8-bit MCU Applications Manual
8-bit MCU
Applications Manual

All products are sold on Motorola's Terms & Conditions of Supply. In ordering a product covered by this document the Customer agrees to be bound by these Terms & Conditions and nothing contained in this document constitutes or forms part of a contract (with the exception of the contents of this Notice). A copy of Motorola's Terms & Conditions of Supply is available on request.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

The Customer should ensure that it has the most up to date version of the document by contacting its local Motorola office. This document supersedes any earlier documentation relating to the products referred to herein. The information contained in this document is current at the date of publication. It may subsequently be updated, revised or withdrawn.

Includes literature available at July 1992
All trademarks recognized.

© MOTOROLA INC.
All Rights Reserved
DL408/D Rev. 1, 1992
Printed in Great Britain by Tevistock Press (Bedford) Ltd. 5000 8/92
Preface

This compilation of Application Notes, Engineering Bulletins, Design Concepts, etc. was originally published by the European Literature Centre of Motorola Ltd. in Milton Keynes, England, and has subsequently gained worldwide acceptance.

Because of the worldwide popularity of the Application Manuals Series it is important for the reader to take note of the following:

The various Application Notes, Engineering Bulletins, Design Concepts, etc. which are included were developed at Design Centres strategically located throughout the global community and many were originally written to support a local need. Whilst the basic concepts of each of the publications included may have broad global applicability, specific Motorola semiconductor parts may be referred to that are currently available for limited distribution in a specific region and may only be supported by the country of origin of the document in which it is referenced.

Also included in the series for completeness and historical significance are documents that may no longer be available individually because obsolete devices are referenced or perhaps, simply, the original document is out of print. Such items are marked in the Table of Contents, Cross Reference, Abstracts and on the first page of the document with the letters 'HI' to indicate that these documents are included for Historical Information only.

All the Application Notes, Engineering Bulletins, Design Concepts, etc. are included to enhance the user's knowledge and understanding of Motorola's products. However, before attempting to design-in a device referenced in this Series, the user should contact the local Motorola supplier or sales office to confirm product availability and if application support is available.

Thank you.
**Other books in this series include:**

<table>
<thead>
<tr>
<th>Code</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL410/D</td>
<td>Power Applications Manual</td>
</tr>
<tr>
<td>DL411/D</td>
<td>Communications Application Manual</td>
</tr>
<tr>
<td>DL412/D</td>
<td>Industrial Control Applications Manual</td>
</tr>
<tr>
<td>DL413/D</td>
<td>Radio, RF and Video Applications Manual</td>
</tr>
<tr>
<td>DL414/D</td>
<td>FET Applications Manual</td>
</tr>
</tbody>
</table>
Contents

Device Cross Reference .......................................................... 1

Abstracts of Applications Documents ........................................... 3

Applications Documents

AN427 MC68HC11 EEPROM Error Correction Algorithms in C ......................................................... 21
AN431 Temperature Measurement and Display Using the MC68HC05B4 and the MC14489 .................. 33
AN432 128K byte Addressing with the MC68HC11 .............................................................................. 49
AN433 TV On-Screen Display Using the MC68HC05T1 ......................................................................... 73
AN434 Serial Bootstrap for the RAM and EEPROM1 of the MC68HC05B6 ......................................... 93
AN436 Error Detection and Correction Routines for M68HC05 Devices Containing EEPROM .......... 105
AN440 MC68HC05B6 and MC68HC05B5 Serial/Parallel Programming Module ..................................... 117
AN441 MC68HC05E0 EPROM Emulator ................................................................................................. 121
AN442 Driving LCDs with M6805 Microprocessors .............................................................................. 153
AN446 MCM2814 Gang-Programmer Using an MC68HC05B6 ................................................................. 169
AN448 “FLOF” Teletext using M6805 Microcontrollers ............................................................................. 181
AN452 Using the MC68HC11K4 Memory Mapping Logic ........................................................................ 217
AN459 A Monitor for the MC68HC05E0 ................................................................................................. 229
AN890 Low Voltage Inhibit (LVI) Capability of the M6805 HCMOS Microcomputer Family .................. 265
AN900 Using the M6805 Family On-Chip 8-Bit A/D Converter ................................................................. 285
AN940 Telephone Dialling Techniques Using the M6805 ......................................................................... 305
AN974 MC68HC11 Floating-Point Package ............................................................................................... 323
AN991 Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers .... 365
AN1010 MC68HC11 EEPROM Programming from a Personal Computer .................................................. 385
AN1050 Designing for Electromagnetic Compatibility (EMC) with HCMOS Microcontrollers ................ 399
AN1055 M6805 16-bit Support Macros .................................................................................................. 423
AN1057 Selecting the Right Microcontroller Unit ....................................................................................... 465
AN1058 Reducing A/D Errors in Microcontroller Applications ................................................................. 473
AN1060 MC68HC11 Bootstrap Mode ......................................................................................................... 485
AN1064 Use of Stack Simplifies M68HC11 Programming ........................................................................ 527
AN1065 Use of the MC68HC68T1 Real-Time Clock with Multiple Time Bases ...................................... 559
AN1066 Interfacing the MC68HC05CS SIOP to an I2C Peripheral ............................................................. 567
AN1067 Pulse Generation and Detection with Microcontroller Units ....................................................... 587
AN1091 Low Skew Clock Drivers and their System Design Considerations ............................................... 613
AN1097 Calibration-Free Pressure Sensor System ................................................................................... 619
AN1102 Interfacing Power MOSFETs to Logic Devices .......................................................................... 625
AN1120 Basic Servo Loop Motor Control Using the MC68HC05B6 MCU .................................................. 635
AN1203 A Software Method for Decoding the Output from the MC14497/MC3373 Combination ............ 643
ANE405 Bi-Directional Data Transfer Between MC68HC11 and MC6805L3 Using SPI* ......................... 649
ANE418 MC68HC05B6 Low-Cost EEPROM Microcomputer Programming Module* .............................. 659
ANE420 Monitor Program for the MC68HC05B6 Microcomputer Unit* ..................................................... 661
EB400 Secure Single Chip Microcomputer Manufacture .......................................................................... 683
EB401 SCAM Modules for Smart Cards ................................................................................................. 691
EB404 “Memories Are Made of This” ... a Look at Memory Considerations for Smart Card Applications ................................................................. 693
EB405 Smart Cards: How to Deal Yourself a Winning Hand .................................................................... 705
EB408 MC68HC705T3 Bootloader ........................................................................................................... 713

Additional Information ................................................................................................................................. 723
Device Cross Reference
## Device Cross Reference

This quick-reference list indicates where specific components are featured in applications documents reproduced in this Manual.

<table>
<thead>
<tr>
<th>Device</th>
<th>Applications Document Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>M68HC05</td>
<td>AN431, AN346, AN442, AN1203</td>
</tr>
<tr>
<td>M68HC05E0</td>
<td>AN459</td>
</tr>
<tr>
<td>M68HC11</td>
<td>AN427, AN432, AN1058, AN1060, AN1064, AN1102, AN1203</td>
</tr>
<tr>
<td>M6805</td>
<td>AN442</td>
</tr>
<tr>
<td>MC68HC05B4</td>
<td>AN431</td>
</tr>
<tr>
<td>MC68HC05B6</td>
<td>AN1097, AN1120, AN348</td>
</tr>
<tr>
<td>MC68HC05C4</td>
<td>AN991, AN1067, AN1055</td>
</tr>
<tr>
<td>MC68HC05C5</td>
<td>AN1066</td>
</tr>
<tr>
<td>MC68HC05E0</td>
<td>AN441</td>
</tr>
<tr>
<td>MC68HC05J1</td>
<td>AN1067</td>
</tr>
<tr>
<td>MC68HC05L6</td>
<td>AN442</td>
</tr>
<tr>
<td>MC68HC05SC11</td>
<td>EB401</td>
</tr>
<tr>
<td>MC68HC05SC21</td>
<td>AN436</td>
</tr>
<tr>
<td>MC68HC05T1</td>
<td>AN343</td>
</tr>
<tr>
<td>MC68HC05T7</td>
<td>AN448</td>
</tr>
<tr>
<td>MC68HC11</td>
<td>AN974, AN1010, AN1066</td>
</tr>
<tr>
<td>MC68HC11A8</td>
<td>AN1067</td>
</tr>
<tr>
<td>MC68HC11A8P1</td>
<td>AN1065</td>
</tr>
<tr>
<td>MC68HC11G5</td>
<td>AN432</td>
</tr>
<tr>
<td>MC68HC11K4</td>
<td>AN452</td>
</tr>
<tr>
<td>MC68HC68T1</td>
<td>AN1065</td>
</tr>
<tr>
<td>MC68HC705B5</td>
<td>AN440</td>
</tr>
<tr>
<td>MC68HC705C8</td>
<td>AN1067</td>
</tr>
<tr>
<td>MC68HC705T3</td>
<td>EB408</td>
</tr>
<tr>
<td>MC68HC805B6</td>
<td>AN440</td>
</tr>
<tr>
<td>MC74LS26</td>
<td>AN1102</td>
</tr>
<tr>
<td>MC3373</td>
<td>AN1203</td>
</tr>
<tr>
<td>MC6805L3</td>
<td>AN348</td>
</tr>
<tr>
<td>MC6805SC01</td>
<td>EB401</td>
</tr>
<tr>
<td>MC6805SC03</td>
<td>EB401</td>
</tr>
<tr>
<td>MC14499</td>
<td>AN431</td>
</tr>
<tr>
<td>MC14497</td>
<td>AN1203</td>
</tr>
<tr>
<td>MC68705P3</td>
<td>AN940</td>
</tr>
<tr>
<td>MC68705R3</td>
<td>AN991</td>
</tr>
<tr>
<td>MC144115</td>
<td>AN441</td>
</tr>
<tr>
<td>MC144115P</td>
<td>AN442</td>
</tr>
<tr>
<td>MC145000</td>
<td>AN442</td>
</tr>
<tr>
<td>MC145003</td>
<td>AN442</td>
</tr>
<tr>
<td>MC145004</td>
<td>AN442</td>
</tr>
<tr>
<td>MCC68HC05SC11</td>
<td>EB400</td>
</tr>
<tr>
<td>MCC68HC05SC21</td>
<td>EB400</td>
</tr>
<tr>
<td>MCM60L256</td>
<td>AN441</td>
</tr>
<tr>
<td>MCM2814</td>
<td>AN446</td>
</tr>
<tr>
<td>MPM3004</td>
<td>AN1120</td>
</tr>
<tr>
<td>MXP2000</td>
<td>AN1097</td>
</tr>
<tr>
<td>MTP3055E</td>
<td>AN1102</td>
</tr>
<tr>
<td>MTP3055EL</td>
<td>AN1102</td>
</tr>
<tr>
<td>PCF8573</td>
<td>AN1066</td>
</tr>
</tbody>
</table>
Abstracts of Applications
Documents
Abstracts

AN427  MC68HC11 EEPROM Error Correction Algorithms in C

A modified Hamming code is used to correct one-bit errors and detect two-bit errors in data blocks of up to 11 bits—avoiding the problem of erroneous correction of two-bit errors. The technique is implemented entirely in 'C', and additional functions are provided to program and read MC68HC11 EEPROM using the encoding/decoding algorithms.

AN431  Temperature Measurement and Display Using the MC68HC05B4 and the MC14489

Shows the basic building blocks of a temperature control system based on the M68HC05 B-series MCUs. Software routines provided include Look-Up Table Interpolation, Binary to BCD Conversion, Degrees C to Degrees F Conversion, and the basis of a real-time counter/clock. Uses a thermistor as the sensing element to allow easy interfacing to the A/D converter of the MC68HC05B4, but the software principles are easily adapted to other sensors.

AN432  128K byte Addressing with the M68HC11

The 64K byte direct addressing capability of the M68HC11 family is insufficient for some applications. This note describes two methods of memory paging—one software-only, the other hardware plus software—that allow the MCU to address a 1Mbit EPROM (128K bytes) by manipulation of the address lines. The two methods illustrate the concept of paging and the inherent compromises; the technique may be expanded to other memory combinations. Includes full software listings.

AN433  TV On-Screen Display Using the MC68HC05T1

The T-series devices in the M68HC05 MCU family provide a convenient and cost-effective means of adding On Screen Display capability (OSD) to TVs and VCRs. The MC68HC05T1 is at the centre of the T-series price/performance range, and is used in this example. Full software listings are provided for a ROM-efficient implementation of an 8-row by 16-character display, including Programme Change, Channel Mode, Automatic Search, Analogues and Channel Name.

AN434  Serial Bootstrap for the RAM and EEPROM of the MC68HC05B6

The MC68HC05B6 has 256 bytes of on-chip EEPROM, called EEPROM1, which can be used for non-volatile data storage. In many applications EEPROM1 stores a look-up table or system set-up variables—in these cases it is necessary to initialise the memory during system manufacture. The RAM bootstrap program in the 'B6 mask ROM uses a simple protocol in order to save ROM space, and cannot accept the S-records that are the normal assembler output. This note explains how to convert assembler output to the 'B6 bootstrap format, and how to bootstrap data into EEPROM1.

AN436  Error Detection and Correction Routines for M68HC05 Devices Containing EEPROM

Applications based on M68HC05 MCUs increasingly require large amounts of critical data to be stored in the on-chip EEPROM. This note describes HC05 software routines which allow stored data to be encoded so that single bit errors in retrieved data may be corrected, and two bit errors detected. The routines use a simple Linear Block Code (Hamming Code) for encoding the stored data. They were written originally for the MC68HC05-SC21 Smart Card MPU, but can be modified easily to run on any HC05 MCU with EEPROM.

AN440  MC68HC805B6 and MC68HC705B5 Serial/Parallel Programming Module

The MC68HC05B6 serial/parallel programmer module allows the user to program MC68HC805B6 and MC68HC705B5 MCUs. This note describes its various operating modes, and gives details of its construction and use. Includes circuit diagram and parts list.

AN441  MC68HC05EO EEPROM Emulator

Unlike other members of the M6805 family, the MC68HC05EO has no on-chip ROM but can address a full 64K bytes of external memory; the external memory may be ROM, EEPROM, RAM and/or additional hardware. This EEPROM emulator illustrates a typical use of this type of MCU; it includes a keyboard, LCD, serial communication and 64K of paged RAM. It can replace with RAM the program ROM or EEPROM in a target system through a cable connection to the system's EEPROM socket, and can be used to debug and modify the target system software. Includes an assembled listing of the emulator control program.

AN442  Driving LCDs with M6805 Microprocessors

The MC68HC05L series of MCUs include circuitry for direct LCD drive. Other MCUs in the M6805 and M68HC05 families have a variety of I/O and display drive capabilities. This comprehensive note describes alternative LCD drive arrangements for applications with different numbers of backplanes and display drive capabilities, including software-based and display driver chip solutions. Circuits and software listings are provided. The techniques apply equally to other MCU families such as the M6801 and M68HC11.

AN446  MCM2814 Gang-Programmer Using an MC68HC805B6

Non-volatile memories (NVM) such as the MCM2814 are widely used in consumer equipment such as television receivers to store semi-permanent, user-defined information. They may also contain data such as optimum sound and picture settings. In a production environment, the initial loading of this data can be achieved quickly by copying an existing NVM. This note describes a programmer based on an MC68HC805B6 which in four
Abstracts (continued)

seconds can fully program eight MCM2814s in parallel and verify them individually.

AN448  "FLOF" Teletext using M6805 Microcontrollers

The "F" members of Motorola's M68HC05 MCU family provide a cost-effective method of adding On Screen Display (OSD) to TVs and VCRs. This note describes an example of Full Level One Feature (FLOF) Teletext control software written for the MC68HC05T7 to control type 5243 Teletext chips. Around 3K bytes of ROM are used, allowing the code to fit with tuning, OSD and stereo functions into the 7.9K bytes of the MC68HC05T7. The example software includes the Spanish implementation of Packet 26; Packet 26 allows for the substitution of specific characters for a particular country.

AN452 Using the MC68HC11K4 Memory Mapping Logic

The MC68HC11K4 includes memory expansion logic which allows the 64 KByte addressing range of the M68HC11 CPU to be extended to more than 1 MByte. This note discusses the operation of this logic and provides examples of memory maps and possible hardware configurations.

AN459 A Monitor for the MC68HC05E0

Development systems for single-chip MCUs can be complex and relatively expensive. This can dissuade potential users from designing them into new applications. This note describes a simple "entry level" development system suitable for debugging hardware and software for the M6805 family of microprocessors. Includes full descriptions, circuit diagram and a listing of the monitor software.

AN890 Low Voltage Inhibit (LVI) Capability of the M6805 HCMOS Microcomputer (MCU) Family

The LVI option provides a cost effective means for the MCU to sense a drop in supply voltage and then shut itself down in well-defined manner. Because the option does not require any additional external parts it provides an overall product cost reduction. The LVI option is provided at the time of manufacture by on-chip circuitry contained in part of the user's ROM pattern. This application note includes an LVI schematic diagram as well as a listing of the monitor and self-check programs.

AN900 Using the M6805 Family On-Chip 8-Bit A/D Converter

Factors which should be considered when using on-chip analog-to-digital (A/D) converters are covered. The pertinent circuit elements and terminology are defined and a self-test hardware/software technique is illustrated. An example on how to manipulate the converted analog data from a temperature sensor is given. It is intended for the digital designer with little or no programming experience.

AN940 Telephone Dialling Techniques Using the MC6805

Intelligent telephones are increasing in popularity – MCUs from the versatile M6805 family make ideal controllers. This demonstration board, based on an MC68705P3 single-chip MCU, shows two cost-effective methods of DTMF and pulse-type dialling. Full hardware schematic and software listings included.

AN974 MC68HC11 Floating-Point Package

While most MC68HC11 applications can be implemented using 16-bit integer precision, certain algorithms may be difficult or impossible without floating-point. This application note details an efficient floating-point package that includes basic trig functions and square root in addition to add, subtract, multiply and divide. It requires just over 2k bytes of memory, with only 10 bytes of page zero RAM in addition to stack RAM.

AN991 Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers

Communication between multiple processors can be difficult when different types are used. One solution is the SPI, an interface intended for communication between ICs on the same board. It can be implemented in software, allowing communication between two MCUs where one has SPI hardware and the other does not. Costly expansion buses and UARTs are eliminated. The scheme is illustrated with a temperature/time display circuit using an MC68HC05C4 and an MC68705R3.

AN1010 MC68HC11 EEPROM Programming from a Personal Computer

Describes a simple and reliable method of programming the MC68HC11's internal EEPROM (or EEPROM connected to its external bus) by downloading data in Motorola S-record format from a standard personal computer (PC) fitted with a serial communications port. Includes basic program for the PC (to Program External EEPROM/RAM, Program Internal EEPROM, or Verify Internal or External EEPROM/RAM) and the source listing of MC68HC11 code for downloading to RAM to receive S records.

AN1050 Designing for Electromagnetic Compatibility (EMC) with HCMOS Microcontrollers

As the operating speeds of the latest HCMOS devices increase, the MCU system designer must take more account of the electromagnetic compatibility (EMC) of the finished product. This discussion relates mainly to emission control, but most of the techniques also reduce electromagnetic susceptibility. Subjects include Legal Requirements, RFI Problems, types of radiation, Supply Decoupling, Grounding Techniques and PCB Layouts. Incorporates an article reprint from EMC Technology describing an EMI/RFI diagnostic probe.
Abstracts (continued)

AN1055 M6805 16-b1t Support Macros
MCUs from the M6805 family are usually chosen for applications requiring small program memory and low computing power, where their low cost is an important benefit. However they may also be used in more advanced applications by employing the advanced software techniques described here. The examples are suitable for 'black box' operation (they may be used without knowing how they work) and consist of macros and subroutines that support pseudo registers on the '6805, simulating registers and addressing modes available on the M68HC11.

AN1057 Selecting the Right Microcontroller Unit
Selecting the proper MCU for an application is one of the critical decisions which can control the success or failure of the project. There are numerous criteria to consider; many of them are presented here along with the thought processes guiding their selection. The reader should attach an appropriate grading scale before evaluating the total and making the correct decision.

AN1058 Reducing A/D Errors in Microcontroller Applications
The MCU with integrated Analogue to Digital Converter provides a highly cost-effective solution for many mixed analogue/digital applications. However, combining a wide bandwidth ADC system on the same die as a high-speed CPU can lead to noise problems in the analogue measurements. This comprehensive note lays down basic system guidelines for the design phase of an MCU-based product, to avoid ADC problems. Includes an examination of a real-world system.

AN1060 M68HC11 Bootstrap Mode
The M68HC11 Bootstrap Mode allows a user program to be loaded into internal RAM through the Serial Communications Interface (SCI). In addition to operating normally, this program can do anything a factory test program can do since the protected control bits become accessible; Expanded Mode resources are available because the control bits can be changed by the bootstrap program. Although the basic concepts are simple, some subtle implications of this mode need careful consideration, both to avoid problems and to find useful applications. Includes commented listings for selected M68HC11 bootstrap ROMs.

AN1064 Use of Stack Simplifies M68HC11 Programming
Architectural extensions to the M6800 family built into the M68HC11 allow easy manipulation of data on the stack. The CPU uses the stack for subroutine and interrupt return addresses. This note discusses two additional uses – the storage of local variables and subroutine parameter passing – that can simplify programming and debugging. It describes the basic operation of the M68HC11 stack, the concept of local and global variables, subroutine parameter passing, and the use of the instruction set to achieve the additional uses. Includes example listings illustrating the techniques.

AN1065 Use of the MC68HC68T1 Real-Time Clock with Multiple Time Bases
The MC68HC68T1 Real Time Clock plus RAM can use a crystal or the 50/60Hz line frequency as its timebase; a Serial Peripheral Interface is provided for communication with a microcomputer. Applications are often line powered during normal operation, using the line frequency as timebase, but must continue to maintain the correct time of day from a crystal source when mains power is lost. The MC68HC68T1 is not capable of switching between the two frequency sources directly, and additional support by the MCU is necessary. This note describes the necessary hardware and software, based on an MC68HC11A8P1 MCU.

AN1066 Interfacing the MC68HC05C5 SIOP to an I^C Peripheral
A standard MCU may not have all the peripherals required in a system on chip. The problem can be solved by interfacing the MCU to off-chip peripherals, ideally using a synchronous serial communication port. Unfortunately these peripherals may not have an interface that is compatible with Motorola's simple synchronous Serial I/O Port (SIOP). This note describes how the SIOP on the MC68HC05C5 can be interfaced to an I^C peripheral, in this case the PCF8573 Clock/Timer. Includes circuit and software listings for a timer/calendar application that can interface with a terminal.

AN1067 Pulse Generation and Detection with Microcontroller Units
MCUs are often required to generate timed output pulses, and to detect and measure input pulses. Output pulses might strobe a display latch, transmit a code or meter an action in a process control system. Input pulses can range from microseconds to hours, and include detecting pushbutton closures, receiving codes or measuring engine rotation. This note describes various methods of generation and detection using several families of Motorola MCUs with differing timer structures. Includes program listings.

AN1091 Low Skew Clock Drivers and their System Design Considerations
With microprocessor-based systems now running at 33MHz and beyond, low-skew clock drivers have become essential – Motorola produces several devices with less than 1ns skew between outputs. Unfortunately, simply plugging one of these high performance clock drivers into a board does not guarantee trouble-free operation. Careful board layout and system noise considerations must also be taken into account.
AN1097  Calibration-Free Pressure Sensor System
The MPX2000 Series of pressure transducers give an output signal proportional to applied pressure. They are available as both ported and unported assemblies for pressure, vacuum and differential measurement. By using the on-chip A/D converter of the MC68HC05B6 MCU, an accurate, reliable and versatile pressure measurement system can be designed which needs no external calibration.

AN1102  Interfacing Power MOSFETs to Logic Devices
Most popular power MOSFETs need 10 volts of gate drive to support their maximum drain current. This creates problems when attempting to drive from 5 volt logic. The new Logic Level power MOSFETs solve some but not all of the problems. This note discusses easy methods of directly interfacing both types of MOSFET to TTL and CMOS logic, and to microprocessors such as the M68HC11. Discusses a method of calculating switching times, to minimise switching losses, and stresses the significance of logic power supply variations.

AN1120  Basic Servo Loop Motor Control Using the MC68HC05B6 MCU
A Proportional Derivative (PD) closed-loop speed control for a brush motor can be created using four integrated circuits, two opto discretes and less than 200 bytes of code. The use of an MCU in feedback control systems is increasingly commonplace. It is justified when system flexibility is needed, for example to accommodate varying drive motors or to allow wear parameters to be stored in EEPROM. This design is based on an MC68HC05B6 MCU and an MPM3004 power MOSFET H-bridge.

AN1203  A Software Method for Decoding the Output from the MC14497/MC3373 Combination
Infrared communication is now widely used as a simple and effective means of remote control over short distances. A variety of encoding methods is used, including the biphasic scheme implemented by the MC14497, a complete building block for IR data transmission. The MC3373 is a companion receiver chip to the MC14497, providing front-end processing to interface a photo detector to a TTL level. This note describes, with software listings for the MC68HC11 and the MC68HC05, the decoding of the data at the output of the MC3373.

ANE405  Bi-Directional Data Transfer Between MC68HC11 and MC6805L3 Using SPI
The powerful Serial Peripheral Interface available on many Motorola MCUs is implemented in 2 forms (the HCMOS families support only Level 1, Level 2 is implemented only on HMOs processors). Both levels communicate easily with each other, but Level 2 has additional capabilities including asynchronous communication. This note describes a method of achieving synchronous communication between levels 1 and 2, and explains the on-chip differences in SPI implementation.

ANE418  MC68HC805B6 Low-Cost EEPROM Microcomputer Programming Module
The EEPROM feature of the MC68HC805B6 microcomputer enables the user to emulate the MC68HC05B6 and the MC68HC05B4. This note describes one programming technique for the MC68HC805B6 internal EEPROM, and describes the design of the simple programming module required.

ANE420  Monitor Program for the MC68HC05B6 Microcomputer Unit
A monitor program is available in the mask ROM of a 68HC05B6 MCU (XC68HC05B6FN MONITOR) which allows the user to write and debug small portions of 68HC05B6 code. It is used in conjunction with a monitor circuit module, +5V power supply and RS-232 terminal. This note includes a description of the facilities available from the software, a circuit diagram of the module and a listing of the monitor code.

EB400  Secure Single Chip Microcomputer Manufacture
Security is the fundamental requirement in designing and manufacturing Smart Card MCUs. This Bulletin summarises the purpose and history of Smart Cards, and explains some of the problems of testing devices after manufacture without prejudicing security. The manufacturing process is necessarily different to that of 'normal' MCUs.

EB401  SCAM Modules for Smart Cards
Motorola's SCAM range of assembly modules consists of the Smart Card product family packaged for insertion in ISO standard plastic cards. This Bulletin lists the planned devices and shows the IS7816/2 contact dimensions, locations and connections. All devices conform to all relevant ISO standards.

EB404  "Memories Are Made of This" ... a Look at Memory Considerations for Smart Card Applications
A Smart Card application typically uses many millions of units per year, so unit cost is crucial to its success. This paper discusses some of the issues concerning memory size and type – and their effect on the specification and cost of secure microcomputers – with particular reference to physical size. (11pp)

EB405  Smart Cards: How to Deal Yourself a Winning Hand
An overview of the current Smart Card market and the various types of product on offer. It looks at ways of determining what features must be provided by a suc-
cessful Smart Card implementation in a given application. Because of the high production volumes, it is essential to choose the optimum product, and to ask the right questions at the start.

**EB408 MC68HC705T3 Bootloader**

This bootloader for the MC68HC705T3 has four switch-selected modes of operation. In addition to programming and verifying the internal EPROM from an external EPROM, it is also possible to load and execute a program in RAM locations $0100-$01FF. A handshake facility is included to allow the external EPROM to be replaced by an intelligent data source and to provide a limited debug capability. Includes circuit diagram and software listing.
Applications
Documents
MC68HC11 EEPROM Error Correction Algorithms in C

By Richard Soja
Motorola Ltd
East Kilbride
Glasgow

INTRODUCTION

This application note describes a technique for correcting one bit errors, and detecting two bit errors, in a block of data ranging from 1 to 11 bits in length. The technique applied is a modified version of a Hamming code, and has been implemented entirely in C. Additional functions have been provided to program and read the EEPROM on an MC68HC11 microcontroller unit using the error encoding and decoding algorithms.

ENCODING AND DECODING ALGORITHMS

Some texts [1], [2] describe the use of simultaneous equations to calculate check bits in Hamming distance-3 error correcting codes. These codes are so named because there are at least 3 bit differences between each valid code in the set of available codes. The codes are relatively easy to generate and can be used to correct one bit errors. However, their main drawback is that if two bit errors occur, then the correction will be made erroneously. This is because the condition of two bit errors corresponds exactly with a one bit error from another valid code.

The technique described here is based on an algorithmic strategy which produces Hamming distance-4 codes over the range of 1 to 11 data bits. This type of code is capable of correcting single bit errors and detecting 2 bit errors.

Alternatively, if the errors are only to be detected, without correction, then up to 3 bit errors can be detected. The reason for this is that the condition of a 3 bit error in one code corresponds to a one bit error from an adjacent valid code. The implication of this is that, if the algorithms are used to correct errors, then a 3 bit error will be corrected erroneously, and flagged as a 1 bit error.

The C program is divided into 3 modules, plus one header file:

1. EECOR1.C
   This is the main program segment, and serves only to illustrate the method of calling and checking the algorithms.

2. HAMMING.C
   This module contains the functions which encode and decode the data.

3. EEPROG.C
   This module contains the EEPROM programming functions tailored for an MC68HC11 MCU.

4. HC11REG.H
   This is the header file which contains the MC68HC11 I/O register names, defined as a C structure.

IMPLEMENTATION OF ERROR CORRECTION STRATEGY

The basic principle of decoding the error correcting codes is to use a Parity check matrix, H, to generate a syndrome word which identifies the error. The H matrix can be generated as follows:

1. Identify how many data bits are needed. For example: 8 data bits

2. Use the standard equation to derive the number of check bits required: if k is the number of check bits, and m the number of data bits, then for the Hamming bound to be satisfied:

   \[ 2^k \geq m + k + 1 \]
A simple way to understand why this equation holds true is as follows: If one can generate a check code which is able to identify where a single error occurs in a bit stream, then the check code must have at least the same number of unique combinations as there are bits in the bit stream, plus 1 extra combination to indicate that no error has occurred. e.g. if the total number of data plus check bits were 7, then the check code must consist of 3 bits, to cover the range 1 to 7 plus one extra (0) to indicate no error at all.

In this example, if \( m=8 \) then, by rearranging the above equation:

\[
2^k - k - 1 \geq S
\]

One way to solve for \( k \) is to just select values of \( k \) starting at say, 1 and evaluating until the bound is reached. This method is implemented algorithmically in function InitEncode() in module HAMMING.C

For \( m=8 \), the solution is \( k=4 \). Note that this value exceeds the Hamming bound, which means that additional data bits can be added to the bit stream, thus increasing the efficiency of the code. In fact, the maximum number of data bits is 11 in this case.

3. A Parity matrix, \( H \) is created from a ‘horizontally orientated’ binary table. The number of columns (\( b_1 \) to \( b_{12} \)) in the matrix correspond to the total number of data and check bits, and the number of rows (\( r_1 \) to \( r_4 \)) to the number of check bits.

\[
\begin{array}{cccccccccccc}
  & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & b_7 & b_8 & b_9 & b_{10} & b_{11} & b_{12} \\
r_1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
r_2 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
r_3 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
r_4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Because the \( H \) matrix in this form, is simply a truncated 4 bit binary table, it can easily be generated algorithmically.

4. The position of all the check bits (\( C_1 \) to \( C_4 \)) within the encoded word is the position of the single 1s in the columns of \( H \). The remaining bits correspond to the data bits (\( D_1 \) to \( D_8 \)).

\[
\begin{array}{cccccccc}
  & C_1 & C_2 & D_1 & C_3 & D_2 & D_3 & D_4 & C_4 & D_5 & D_6 & D_7 & D_8 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
2 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

5. Each check bit is generated by taking each row of \( H \) in turn, and modulo-2 adding all bits with a 1 in them except the check bit positions.

\[
\begin{align*}
 i.e. & \quad C_1 = D_1 + D_2 + D_4 + D_5 + D_7 \\
 & \quad C_2 = D_1 + D_3 + D_4 + D_6 + D_7 \\
 & \quad C_3 = D_2 + D_3 + D_4 + D_8 \\
 & \quad C_4 = D_5 + D_6 + D_7 + D_8 \\
\end{align*}
\]

6. The syndrome, \( s \), is the binary weighted value of all check bits.

\[
\text{i.e. } s = 1 \cdot C_1 + 2 \cdot C_2 + 4 \cdot C_3 + 8 \cdot C_4
\]

7. The error position (i.e. column) is determined by the value of the syndrome word, provided it is not zero. A zero syndrome means no error has occurred. Note that this error correction technique can correct errors in either data or check bits - which is not necessarily the case with certain other error correction strategies.

The advantage of this method, where the check bits are interspersed in a binary manner throughout the code word, is that the error position can be calculated algorithmically.

An important point to note is that the parity check matrix described above generates Hamming distance-3 codes, which means that 2 errors will cause erroneous correction. This can be fixed by adding an extra parity check bit, \( C_5 \), which is the modulo-2 addition of all data and check bits together.

\[
\text{i.e. } C_5 = C_1 + C_2 + D_1 + C_3 + D_2 + D_3 + D_4 + C_4 + D_5 + D_6 + D_7 + D_8
\]

The code word then becomes:

\[
C_1 \quad C_2 \quad D_1 \quad C_3 \quad D_2 \quad D_3 \quad D_4 \quad C_4 \quad D_5 \quad D_6 \quad D_7 \quad D_8 \quad C_5
\]

To determine if an uncorrectable error has occurred (i.e. 2 errors) in the received word, the extra parity bit is tested. If the syndrome is non-zero and the parity bit is wrong, then a correctable error has occurred. If the syndrome is non-zero and the parity bit is correct, then an uncorrectable error has occurred.

**EFFICIENCY**

The following table lists the relative efficiencies of this algorithm, against data size.

<table>
<thead>
<tr>
<th>Data bits</th>
<th>Encoded bits</th>
<th>Efficiency %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>64</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>67</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>69</td>
</tr>
</tbody>
</table>

22
The implementation of the above techniques are given in the module HAMMING.C.

In order to maintain orthogonality in the EEPROM algorithms, the encoded data used by the functions in module EEPROG.C are forced to either 1 byte or 2 byte (word) sizes. This also eliminates the complexities of packing and unpacking data in partially filled bytes.

CONCLUSIONS

In this application note, the encoding algorithm’s generator matrix is the same as the parity check matrix.

The C functions <read> and <write> in the module HAMMING.C return a status value - 0, 1 or 2 - which indicates whether the data has no errors, 1 corrected error, or 2 erroneously corrected errors. This means that if the status value is 0 or 1, then the data can be assumed good. If the status value is 2, then the data will be bad.

Alternatively the functions can be used for error detection only, without correction. In this case, a status value of 1 corresponds to either 1 or 3 bit errors, while a status value of 2 indicates that 2 bit errors have occurred.

By using the C functions listed in this application note, the encoded data size can easily be changed dynamically. To do this, the function <InitEncode> must be called with the required new data size. The global variables used by all the encoding, decoding and EEPROM programming and reading functions are automatically updated. This allows the encoding and error correction process to be virtually transparent to the user. In addition, the functions <write> and <read> will automatically increment the address pointer by the correct encoded data size set up by <InitEncode>. This simplifies the structure of loops to program and read back data. Example code is provided in module EECOR1.C.

The encoding and decoding algorithms listed here may be applied to other forms of data, such as that used in serial communications, or for parallel data transfers.

By incorporating the error correction or detection-only schemes described in this application note, the integrity of data storage and transfer can be greatly improved. The impact on EEPROM usage is to increase its effective reliability and extend its useful life beyond the manufacturers’ guaranteed specifications.

REFERENCES


MODULE EECOR1.C

Tests EEPROM error detection using a modified hamming encoding scheme.

typedef unsigned char byte;
typedef unsigned int word;

/* Global variables used by main() */
byte *ee_addr,*start_addr,*end_addr,i,Error;
word data;

/***************************************************************************/
/* External global variables */
extern byte CodeSize; /* number of bits in encoded data */

/***************************************************************************/
/* External Functions */
extern byte read(word *data,byte **addr); /* Function returns error status */
extern byte write(word data,byte **addr); /* */
/* Table of Status returned by read and write functions */
Returned Status Condition
0 No errors detected or corrected.
1 One error detected and corrected.
2 Two errors detected, but correction is erroneous.

Notes:
1/ When the returned value is 2, the function <read> will returned a bad value in variable <data> due to the inability to correctly correct two errors. <read> also automatically increments the address pointer passed to it, to the next memory space. The incremented value takes into account the actual size of the encoded data. i.e. either 1 or 2 byte increment.

2/ Function <write> also performs a read to update and return an error status. This gives an immediate indication of whether the write was successful. <write> also automatically increments the address pointer passed to it, to the next free memory space. The incremented value takes into account the actual size of the encoded data. i.e. either 1 or 2 byte increment.

int main()
{
    CodeSize=InitEncode(11); /* Get code size (less 1) needed by 11 data bits */
    */* by 11 data bits */
    ee_addr=(byte *)0xb600; /* Initialise EEPROM start address */
    for(i=1;i<0x10;i++) /* and 'erase' EEPROM */
        Error=write(0x7ff,ee_addr); /* Function successful if Error<>2 */
    ee_addr=(byte *)0xb600; /* Reset EEPROM address */
    Error=write(0x5aa,ee_addr); /* Write 0x5aa & increment ee_addr */
    Error=write(0x255,ee_addr); /* Write 0x255 at next available address */
    CodeSize=InitEncode(4); /* Change number of data bits to 4 */
    start_addr=ee_addr; /* Save start address for this data */
    for(i=1;i<0x10;i<<1) /* Program 'walking 1's' */
        Error=write(i,ee_addr);
    end_addr=ee_addr; /* Save end address */
    ee_addr=start_addr;
    while (ee_addr<end_addr) /* Read back all the 4 bit data */
        Error=read(data,ee_addr); /* <data> good if Error=0 or 1 */
} /* main */
**MODULE HAMMING.C**

/* Modules to Generate hamming codes of distance 4, for data sizes in the range 1 bit to 11 bits. The upper bound is limited by the encoded word type bit range (16 bits).

Corrects 1 bit error in any position (check or data), and detects 2 bit errors in any position.

After execution of the <Decode> function, the global variable <ErrFlag> is updated to indicate level of error correction.

i.e. ErrFlag Condition
    0 No errors detected or corrected.
    1 One error detected and corrected.
    2 Two errors detected, but correction is erroneous.

Note that when ErrFlag is 2, function <Decode> will return a bad value, due to its inability to correctly correct two errors.
*/

#define TRUE 1
#define FALSE 0
typedef unsigned char byte;
typedef unsigned int word;

byte DataSize, CodeSize, EncodedWord, ErrFlag;

/* Function prototypes */
byte OddParity(word Code);
word Power2(byte e);
byte InitEncode(byte DataLength);
word MakeCheck(word Data);
word Encode(word Data);
word Decode(word Code);

byte OddParity(Code)
word Code;
/*
Returns TRUE if Code is odd parity, otherwise returns FALSE */
{
    byte p;
    p=TRUE;
    while (Code!=0)
    {
        if (Code & 1) p=!p;
        Code>>=1;
    }
    return(p);
}

word Power2(e)
byte e;
/*
Returns 2^e */
{
    word P2;
    signed char i;
    P2=1;
    if ((signed char)(e)<0)
        return(0);
    else
{  for (i=1;i<=(signed char)(e);i++)
      P2<<=1;
  return(P2);
}

byte InitEncode(DataLength)
byte DataLength;
/*
Returns the minimum number of total bits needed to provide
Hamming distance 3 codes from a data size defined by passed
variable <DataLength>. This value also updates global variable <DataSize>.
i.e. finds the minimum solution of (k+m) for the inequality:
\[ 2^k \geq k + m + 1 \]
In addition, updates global variable <EncodedSize> to reflect number of bytes
per encoded data. <EncodedSize> will be either 0 or 1.
*/

{  byte CheckLength,i;
  DataSize=DataLength; /* DataSize used by other functions in this module */
  CheckLength=1;
  while (((Power2(CheckLength)-CheckLength-1)<DataLength)
          CheckLength++;
  i=CheckLength+DataLength;
  EncodedWord=i / 8;
  /* =0 if byte sized, =1 if word sized */
  return (EncodedWord+DataLength);
}

word MakeCheck(Data)
word Data;
/*
Returns a check word for Data, based on global variables <DataSize>
and <CheckSize>. The H parity matrix is generated by a simple for loop.
*/

{  byte i,H,CheckSize,CheckValue,Check,CheckMask;
  word DataMask;
  
  Check=0;
  CheckMask=1;
  CheckSize=CodeSize-DataSize;
  for (i=1;i<=CheckSize;i++)
  {  
    CheckValue=FALSE;
    DataMask=1;
    for (H=1;H<=CodeSize;H++)
    {  
      if ((0x8000 & H)!=0) /* Column with single bit set */
      {  
        if (((H & CheckMask)!=0)
            CheckValue=((DataMask & Data)!=0); 
          DataMask<<=1;
        }
      }
      if (CheckValue) Check|=CheckMask;
      CheckMask<<=1;
    }
  return (Check);  
}
word Encode(Data)
word Data;
/*
Returns an encoded word, consisting of the check bits concatenated on to the most significant bit of <Data>. A single odd parity bit is concatenated on to the Encoded word to increase the hamming bound from 3 to 4, and provide 2 bit error detection as well as 1 bit correction. Uses global variables <Datasize> and <CodeSize> to determine the concatenating positions.
*/

{ word Code;
  Code=Data | (MakeCheck(Data)<<DataSize);
  if (OddParity(Code))
    Code|=Power2(CodeSize);
  return(Code);
}

word Decode(Code)
word Code;
/*
Returns the error corrected data word, decoded from <Code>. Uses global variable <Data Size> to determine position of the check bits in <Code>. Updates global variable <ErrFlag> to indicate error status i.e.:

<table>
<thead>
<tr>
<th>ErrFlag</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors found</td>
</tr>
<tr>
<td>1</td>
<td>Single error corrected</td>
</tr>
<tr>
<td>2</td>
<td>Double error - invalid correction</td>
</tr>
</tbody>
</table>
*/

{ word ParityBit,Data,Check,ErrorCheck,Syndrome,DataMask;
  byte DataPos,CheckSize,CheckPos,H,DataBit;

  ErrFlag=0;
  ParityBit=Code & Power2(CodeSize);  /* Extract parity bit. */
  DataMask=Power2(DataSize)-1;       /* Make data mask */
  Data|=Code & DataMask;             /* Extract data bits. */
  CheckSize=CodeSize-DataSize;       /* Extract check bits, */
  Check=(Code>>DataSize) & (Power2(CheckSize)-1);  /* ignoring parity. */
  ErrorCheck=MakeCheck(Data);
  Syndrome=Check ^ ErrorCheck;       /* Get bit position of error. */
  if (Syndrome>0) ErrFlag++;         /* Increment flag if error exists. */
  H=0;
  DataPos=0;
  CheckPos=DataSize;
  DataBit=TRUE;

  while ((H!=Syndrome) & (DataPos<DataSize))  /* Identify which data or*/
  { H++;
    DataBit=(0x8000 % H);
    if (DataBit) DataPos++;
    else CheckPos++;
  }
  if (DataBit) Code|=Power2(DataPos-1);
  else Code|=Power2(CheckPos-1);
  Code|=ParityBit;
  if (OddParity(Code)) ErrFlag++;
  return(Code & DataMask);
}
MODULE EEPROG.C

/*Module to program MC68HC11 EEPROM.

Contains <read> and <write> functions to encode and decode data
formatted by modified hamming scheme.

* /

#include <HC11REG.H>
define regbase (*(struct HC11IO *) 0x1000)
define eras 0x16
define writ 0x02
typedef unsigned char byte;
typedef unsigned int word;

union twobytes
{
  word w;
  byte b[2];
} udata;

extern byte EncodedWord,ErrFlag;

/* Function prototypes */

extern word Encode(word Data);
extern word Decode(word Code);

void delay(word count);
void eeprog(byte val,byte byt,byte *addr,word count);
void program(byte byt,byte *addr);
byte read(word *data,byte **addr);
byte write(word data,byte **addr);

void delay(count)
word count;
{
  regbase.TOC1.regbase.TCNT+count;
  regbase.TFLG1=0x80;
  do;while ((regbase.TFLG1 & 0x80)==0);
}

void eeprog(val,byt,addr,count)
byte val;
byte byt;
byte *addr;
word count;
{
  regbase.PPROG=val;
  *addr=byt;
  ++regbase.PPROG;
  if (count<100) count=100;
  delay(count);
  --regbase.PPROG;
  regbase.PPROG=0;
}
void program(byt, addr)
byte byt;
byte *addr;
{
eeproq(eras, byt, addr, 20000);
eeproq(writ, byt, addr, 20000);
} /* First erase byte */
/* Then write value */

byte read(data, addr)
word *data;
byte **addr;
{
  udata.b[1] = (*addr)++;
  if (EncodedWord)
    udata.b[0] = (*addr)++;
  else
    udata.b[0] = 0;
  *data = Decode(udata.w);
  return(ErrFlag);
} /* Read back data LSB first, and inc address */
/* If word stored then read MSB */
/* Inc address for next call to this function */
/* else only byte stored, so clear MSB */
/* Decode data, which updates <ErrFlag>, */
/* and return ErrFlag */

byte write(data, addr)
word data;
byte **addr;
{
  byte *oldaddr;

  udata.w = Encode(data);
  oldaddr = addr;
  program(udata.b[1], (*addr)++);
  if (EncodedWord)
    program(udata.b[0], (*addr)++);
  return(read(&udata.w, &oldaddr));
} /* Encode data. */
/* Save initial address for verification. */
/* Program LSB first to allow for either */
/* 1 or 2 byte encoded data */
/* MSB of word sized data, & inc address */
/* Return <ErrFlag> to calling segment */

29
/* HCll structure - I/O registers for MC68HCll */

struct HCllIO {
    unsigned char PORTA; /* Port A - 3 input only, 5 output only */
    unsigned char PORTC; /* Parallel I/O control */
    unsigned char PORTB; /* Port B - Output only */
    unsigned char PORTCL; /* Alternate port C latch */
    unsigned char Reserved1; /* Data direction for port C */
    unsigned char DORD; /* Port D */
    unsigned char PORTE; /* Port E */

    /* Timer Section */
    unsigned char CFORC; /* Compare force */
    unsigned char OClM; /* OCl mask */
    unsigned char OClO; /* OCl data */
    int TCNT; /* Timer counter */
    int TICl; /* Input capture 1 */
    int TIC2; /* Input capture 2 */
    int TIC3; /* Input capture 3 */
    int TOCl; /* Output compare 1 */
    int TOC2; /* Output compare 2 */
    int TOC3; /* Output compare 3 */
    int TOC4; /* Output compare 4 */
    int TOC5; /* Output compare 5 */
    unsigned char TCLl1; /* Timer control register 1 */
    unsigned char TCLl2; /* Timer control register 2 */
    unsigned char TMSKl; /* Main timer interrupt mask 1 */
    unsigned char TFLGl; /* Main timer interrupt flag 1 */
    unsigned char TMSK2; /* Main timer interrupt mask 2 */
    unsigned char TFLG2; /* Main timer interrupt flag 2 */

    /* Pulse Accumulator Timer Control */
    unsigned char PACTL; /* Pulse Acc control */
    unsigned char PACNT; /* Pulse Acc count */
/* SPI registers */
unsigned char SPCR;  /* SPI control register */
unsigned char SPSR;  /* SPI status register */
unsigned char SPDR;  /* SPI data register */

/* SCI registers */
unsigned char BAUD;  /* SCI baud rate control */
unsigned char SCCR1; /* SCI control register 1 */
unsigned char SCCR2; /* SCI control register 2 */
unsigned char SCSR;  /* SCI status register */
unsigned char SCDR;  /* SCI data register */

/* A to D registers */
unsigned char ADCTL; /* AD control register */
unsigned char ADR[4]; /* Array of AD result registers */

/* Define each result register */
#define adrl ADR[0]
#define adr2 ADR[1]
#define adr3 ADR[2]
#define adr4 ADR[3]

unsigned char Rsvr[4]; /* Reserved for A to D expansion */

/* System Configuration */
unsigned char OPTION; /* System configuration options */
unsigned char COPRST; /* Arm/Reset COP timer circuitry */
unsigned char PPREG;  /* EEPROM programming control reg */
unsigned char HPRIO;  /* Highest priority i-bit int & misc */
unsigned char INIT;   /* RAM - I/O mapping register */
unsigned char TEST1;  /* Factory TEST control register */
unsigned char CONFIG; /* EEPROM cell - COP, ROM, & EEPROM en */

};

/* End of structure HCII */
Temperature measurement and display using the MC68HC05B4 and the MC14489

By Jeff Wright, Motorola Ltd., East Kilbride

INTRODUCTION

This application note is intended to show the basic building blocks of a temperature control system based on the MC68HC05Bx family of MCUs. Software routines in the application include look-up table interpolation, binary to BCD conversion, DegC to DegF conversion and the basis of a real time counter/clock. For temperature display the Multi-character LED display driver MC14489 is used, driven from the B4's SCI, resulting in simple hardware with a low component count. The temperature sensing element used here is a thermistor to allow easy interfacing to the A/D converter of the HC05B4, but the software principles shown would be the same for many other types of sensors. A software listing is included at the end of this application note.

TEMPERATURE MEASUREMENT

A pre-calibrated thermistor was chosen as the temperature sensing element. Its characteristic curve over the temperature range of -40 to 80°C is shown in Figure 1. To get the best accuracy from the HC05B4's on-board A/D, the input signal should be scaled to use as much of the available VRH-VRL range as possible. Here VRH is connected to Vdd and VRL is tied to Vss. In this case, using the thermistor as potential divider with a 20kΩ resistor results in a signal range of approximately 0.3V to 4.7V over the -40 to 80°C temperature range. The voltage across the thermistor (input to the A/D), plotted against temperature, is shown in Figure 2.

![Graph of Thermistor Resistance vs Temperature](image)

**Figure 1. Thermistor resistance vs Temperature**
As can be seen from Figure 2, the response is non-linear and so a look-up table approach is the simplest way of obtaining the required accuracy. The thermistor characteristics are stored as a series of points in a table in ROM and a linear interpolation between adjacent points is used to obtain the temperature that corresponds to a given A/D reading. The number of points that must be stored depends on how non-linear the response is and the required accuracy of the result. In this case 16 points were chosen; in order to keep the software simple (and therefore fast), they are spread at intervals of 16 through the A/D result range of 0-255. For each point (16, 32, 48 etc.), the voltage on the A/D input was calculated and the corresponding temperature was obtained from the graph of Figure 2. These points were then used to form the look-up table shown in Figure 3, resulting in a temperature range of -40 to 79°C. Figure 4 shows the reconstructed response of the thermistor obtained by linear interpolation of the points in the look-up table.

![Graph of A/D input voltage vs Temperature](image)

**Figure 2. A/D input voltage vs Temperature (inset: circuit used)**

<table>
<thead>
<tr>
<th>A/D RESULT</th>
<th>A/D (volts)</th>
<th>TEMP (°C)</th>
<th>TEMP (°C 2s Compl)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>0.31</td>
<td>79</td>
<td>4F</td>
</tr>
<tr>
<td>32</td>
<td>0.63</td>
<td>56</td>
<td>38</td>
</tr>
<tr>
<td>48</td>
<td>0.94</td>
<td>43</td>
<td>2B</td>
</tr>
<tr>
<td>64</td>
<td>1.26</td>
<td>34</td>
<td>22</td>
</tr>
<tr>
<td>80</td>
<td>1.57</td>
<td>27</td>
<td>1B</td>
</tr>
<tr>
<td>96</td>
<td>1.88</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>112</td>
<td>2.20</td>
<td>15</td>
<td>0F</td>
</tr>
<tr>
<td>128</td>
<td>2.51</td>
<td>10</td>
<td>0A</td>
</tr>
<tr>
<td>144</td>
<td>2.82</td>
<td>5</td>
<td>05</td>
</tr>
<tr>
<td>160</td>
<td>3.14</td>
<td>-1</td>
<td>FF</td>
</tr>
<tr>
<td>176</td>
<td>3.45</td>
<td>-6</td>
<td>FA</td>
</tr>
<tr>
<td>192</td>
<td>3.77</td>
<td>-11</td>
<td>F5</td>
</tr>
<tr>
<td>208</td>
<td>4.08</td>
<td>-18</td>
<td>EE</td>
</tr>
<tr>
<td>224</td>
<td>4.39</td>
<td>-26</td>
<td>E6</td>
</tr>
<tr>
<td>240</td>
<td>4.71</td>
<td>-40</td>
<td>D8</td>
</tr>
<tr>
<td>255</td>
<td>5.0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Figure 3. Interpolated A/D input voltage vs Temperature**
The temperature reading is updated every second; the software to accomplish this is relatively simple:

The timer is set to overflow every 125 mS with a 4.1934 MHz crystal. The timer overflow interrupt routine updates the real time counters TICKS, SECS, MINS & HRS and sets the flag bit SEC every time a second has elapsed.

The main program loop is executed every second (via the SEC flag bit) and after checking the metric/imperial selector switch the temperature is measured by the subroutine ADCONV. This routine starts by reading the thermistor selector switch and setting up the A/D control register accordingly. An A/D conversion is then carried out four times on the selected channel and the results accumulated in the accumulator and the temporary register TEMP. This result is then divided by 4 by rotating, to obtain the average A/D result. The averaging technique is employed to try and reduce the effect of noise on the A/D input. The number of conversions to average is determined by time constraints and the noise levels in the surrounding environment. The upper nibble of the result is then used to access the look-up table to obtain the 'base' temperature value. If the temperature limit is exceeded then the TLIMIT flag is set before exiting from the routine.

Temperature table entries are stored in 2's complement form so that the interpolation between positive and negative values will work successfully. The interpolation is carried out by obtaining the difference between the base value and the next in the table, multiplying this by the lower nibble of the A/D result and then dividing by 16. This result is then subtracted from the base value to obtain the real temperature in 2's complement °C which is stored in the register NEWTMP before exiting from the routine. The difference information is subtracted from the base value rather than added because the thermistor has a negative temperature co-efficient (NTC) so that an increase in the A/D result corresponds to a drop in temperature.

If the imperial mode is selected (°F) then the next stage before updating the display is to convert from °C to °F and this is carried out in the subroutine CTOF.

Converting from °C to °F is accomplished by multiplying by 1.8 and adding 32. First the sign of the temperature in °C is stored via the flag bit NEGNUM, then the maximum °F limit (53 °C) is checked before the magnitude is multiplied by 1.8 (multiply by 115 and divide by 64). Again, use is made of rotating to do the dividing, in order to increase execution speed. The sign of the result is then restored and 32 added to obtain the temperature in 2's complement °F.
TEMPERATURE DISPLAY

An MC14489 multi-character display driver was chosen for this purpose as it can be easily interfaced to a wide range of Motorola MCUs, requires almost no external components and has a character set that includes the degree symbol (°). The MC14489 can also be cascaded if the application was expanded to require a larger display. The MC14489 would normally be driven from an SPI on the MCU but here, since the the 68HC05B family does not have an SPI, use is made of the SCI clock output feature that is available on this family.

Before the temperature can be written to the display driver it has to be converted into the correct data format.

The first stage of this is to convert from 2's complement binary to BCD. This is carried out in the routine CONBCD which is called from SETDISP. The sign of the temperature is stored in the flag bit NEGNUM before SETDISP is called; then, after first checking if the TLIMIT flag is set, the temperature is converted to BCD in DEC() by CONBCD. This is accomplished by rotating left the binary number followed immediately by a rotate left of the BCD result; this has the effect of multiplying the current BCD result by 2 and adding in the new binary bit at the same time. After each rotate the BCD registers are checked and adjusted for overflow (>90) before the bit counter contained in the index register is decremented. This process of rotate then adjust is continued until all the binary bits have been used; the BCD result will then be resident in the registers DECO, 1 & 2.

The rest of the routine SETDISP is concerned with setting up the display registers DISP1, 2, 3 and the display control register DISPC. The MC14489 data format is msb first whereas the 68HC05B4 SCI transmits lsb first; this means that the bit order of the data stream has to be stored in reverse in the display registers. This can be confusing when trying to work out the codes that have to be stored in the B4 to generate a specific character.

Figures 5a and 5b show the 14489 data format and the corresponding bit positions in the B4 registers DISP1, 2, 3 & C. The sign of the temperature is restored and the numeric display registers are configured to display ‘-’ if the temperature limit has been exceeded before exiting from the SETDISP routine.

The main program loop then calls the subroutine DISPL which actually transmits the contents of the display registers to the MC14489 via the SCI. The MC14489 contains special Bit Grabber circuitry that allows either the internal display registers or the configuration register to be updated without address or steering bits so that updating the display involves a simple transmission of either 3 bytes for the display registers or 1 byte for the configuration register. Even for cascaded 14489s there is no need for address bits – see the MC14489 data sheet for more details.

The MC14489 can be clocked at up to 4 MHz at 5 volts so here the maximum transmit baud rate of the SCI is used – 131.072 KHz with a 4.19304 MHz crystal. The transmission of the display data only takes place if there has been a change in the data since the last time. If there has been a change, the 3 data registers are transmitted in turn starting with DISP3 and the OLD registers are updated ready for the change check next time round. After the last byte has gone, the SCI and 14489 are disabled before returning to the main loop.

The last subroutine called from the main program is the 14489 configuration update routine DISCON. This routine operates in a similar manner to DISPL, checking to see if there has been a change to the config. data before transmitting it.

This completes the operation of the program which now jumps back to the start of the main loop and waits for the SEC bit to be set again before repeating the temperature measurement and display sequence.

<table>
<thead>
<tr>
<th>msb</th>
<th>lsb</th>
</tr>
</thead>
<tbody>
<tr>
<td>C7</td>
<td>C6</td>
</tr>
<tr>
<td>C5</td>
<td>C4</td>
</tr>
<tr>
<td>C3</td>
<td>C2</td>
</tr>
<tr>
<td>C1</td>
<td>C0</td>
</tr>
</tbody>
</table>

MC14489 configuration data

<table>
<thead>
<tr>
<th>msb</th>
<th>lsb</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC0</td>
<td>DC1</td>
</tr>
<tr>
<td>DC2</td>
<td>DC3</td>
</tr>
<tr>
<td>DC4</td>
<td>DC5</td>
</tr>
<tr>
<td>DC6</td>
<td>DC7</td>
</tr>
</tbody>
</table>

MC68HC05B4 display register DISPC

Figure 5a. MC14489 to MC68HC05B4 display register mapping
As already mentioned, the use of the MC14489 results in a very low component count for the application; the hardware schematic can be seen in Figure 6. The only I/O pins required are for reading the option switches and for controlling the enable of the MC14489. Pull-downs are required on the clock and data pins as these become high impedance when the SCI is disabled. The LED displays are common cathode; a single external resistor is all that is required to set the brightness level of the displays. In this case though, a light dependent resistor, R12 (ORP12), has been used to control the display brightness for a variety of background lighting conditions. The resistance of R12 decreases with increasing light and so R11 must be incorporated to ensure that the maximum source current spec. of the MC14489 is not exceeded in very bright lighting conditions. R13 ensures there is still enough drive current for the LEDs in dark conditions.

APPLICATION AREAS

As mentioned in the introduction, this application note is designed only to show some fundamental building blocks of a temperature control system based on the 68HC05Bx family of MCUs. Where possible, the software has been written in a modular fashion, so that the routines can easily be transported to another application and the binary to BCD routine could be expanded to handle larger numbers. The large number of I/O, PWMs and timer functions unused show that the 68HC05B family has plenty of functionality left to perform other control functions. For example, in process control, fluid flow or speed sensors could be connected to the timer input capture pins, pressure sensors to the other A/D pins, a keypad to the I/O lines and the other I/O & PWMs used to perform output control functions.
Figure 6. Hardware schematic
\'\'\'\n68HC05B4 TEMPERATURE MEASUREMENT & DISPLAY \'\'\n\'\n\' Jeff Wright, Motorola East Kilbride. Last Updated 22/02/90 \'\n\'\n\' This software was written by Motorola for demonstration \n\' purposes only. Motorola does not assume any liability arising \n\' out of the application or use of this software and does not \n\' guarantee its functionality \n\'\n\'\n
********** I/O and INTERNAL registers definition **********

* *
* I/O registers
*
PORTA EQU $00 port A.
PORTB EQU $01 port B.
PORTC EQU $02 port C.
PORTD EQU $03 port D.
DDRA EQU $04 port A DDR.
DDRB EQU $05 port B DDR.
DDRC EQU $06 port C DDR.
*
* A/D registers
*
ADDATA EQU $08 A/D data register.
ADSTC EQU $09 A/D status and control register.
COCO EQU 7 Conversion complete flag.
*
* SCI registers
*
BAUD EQU $00 SCI baud register.
SCCR1 EQU $01 SCI control register 1.
SCCR2 EQU $02 SCI control register 2.
SCSR EQU $03 SCI status register.
TDR EQU 7
TC EQU 6
SCDAT EQU $11 SCI data register.
*
* TIMER registers
*
TCR EQU $12 Timer control register.
TOIE EQU 5 Timer overflow interrupt enable.
OCIE EQU 6 Timer output compares interrupt enable.
ICIE EQU 7 Timer input captures interrupt enable.
TSR EQU $13 Timer status register.
OCF2 EQU 3 Timer output compare 2 flag.
ICF2 EQU 4 Timer input capture 2 flag.
TOF EQU 5 Timer overflow flag.
OCF1 EQU 6 Timer output compare 1 flag.
ICF1 EQU 7 Timer input capture 1 flag.
65 00000014 TIC1HI EQU $14 Timer input capture register 1 (16-bit).
66 00000015 TIC1LO EQU $15 Timer output compare register 1 (16-bit).
67 00000016 TOC1HI EQU $16 Timer free running counter (16-bit).
68 00000018 TIMHI EQU $18 Timer alternate counter register (16-bit).
69 0000001a TIMAIHI EQU $1A Timer input capture register 2 (16-bit).
70 0000001b TIMALO EQU $1B Timer output compare register 2 (16-bit).
71 0000001c TIMAHI EQU $1C Timer input capture register 2 (16-bit).
72 0000001d TIMAILO EQU $1D Timer output compare register 2 (16-bit).
73 0000001f TIMALO EQU $1F

* MEMORY MAP DEFINITION
*
* TEST EQU $20 TEST register
84 00000020 TEST EQU $20 Start address of ROM.
85 00000020 ROMO EQU $020 Start address of ROM.
86 00000050 RAM EQU $050 Start address of RAM.
87 00000F00 UROM EQU $0F00 Start address of main user ROM.
88 *

*************** RAM ALLOCATION **********************

SECTION .RAM, ADDR=$50

97 00000050 TICKS RMB 1
98 00000051 SECS RMB 1
99 00000052 MINS RMB 1
100 00000053 HRS RMB 1
101 102 00000054 FLAG RMB 1
103 00000000 OVERFL EQU 0
104 00000001 NEGNUM EQU 1
105 00000002 TLIMIT EQU 2
106 00000003 SEC EQU 3
107 108 00000055 MODE RMB 1
109 00000000 IMP EQU 0
110 111 00000056 BIN0 RMB 1
112 00000057 DEC2 RMB 1
113 00000058 DEC1 RMB 1
114 00000059 DEC0 RMB 1
115 116 0000005a NEWIMP RMB 1
117 0000005b TEMP RMB 1
118 0000005c TEMP1 RMB 1
119 0000005d TEMP2 RMB 1
120 121 0000005e DISP1 RMB 1
122 0000005f DISP2 RMB 1
123 00000060 DISP3 RMB 1
124 00000061 DISP4 RMB 1
125 00000062 OLDD1 RMB 1
126 00000063 OLDD2 RMB 1
RESET EQU *  
LDA #$50 Initialise Ports.

STA PORTA

STA PORTB

STA DDRA

STA DDRB

LDX #OLDDC

INIRAM STA ,X Initialise all used RAM locations.

DECX

CPX #RAM

BNE INIRAM

LDA #504

STA PORTC

LDA #504 PC2 output high.

STA DDRC

LDA TSRO clr any pending flags.

LDA TIMLO

LDA #$20 Enable timer overflow

STA TCR interrupt.

CLI

* START OF MAIN PROGRAM LOOP *

MAINLUP EQU *

BRCLR SEC,FLAG,MAINLUP

BCLR SEC,FLAG

BCLR IMP,MODE Check metric/imperial selector.

BCLR 0,PORTC,NOIMP Check degC/degF switch.

BSET IMP,MODE

NOIMP BCLR NEGNUM,FLAG Clear sign indicator.

JSR ADCONV Go measure temperature

JSR IMP,MODE,GOMETR (in degC - 2s compl)

JSR CTOF Convert to degF.

GOMETR LDA NEWTMP

BPL GOMORE

NEGA Only use magnitude to do BCD conv.

BSET NEGNUM,FLAG Remember the sign of the number.

GOMORE STA BING Store temperature for conv to BCD.

JSR SEDISP Set-up display bytes.

JSR DISPL Update display if neccessary.

JSR DISCON Update 14489 config if neccessary.

BRA MAINLUP
*                           CTOF - Converts NEWTMP from degC to degF *
*                           ================================================*

CTOF EQU *

LDA NEWTMP
BPL NONEG
BSET NEGNUM,FLAG

Remember if No is negative or not.

BRA MUL1P8

NONEG CMP $53

Check for max degF limit of 127F.

BSET TLIMIT,FLAG

Set limit and return if over range.

MUL1P8 LDX #115

Multiply by 115 and divide by 64.

(Remember if No is negative or not.)

NEWTMP

MUL RORX

(same as multiplying by 1.8)

BCLR NEGNUM,FLAG, NONEG1

Return sign of number.

SETDISP EQU *

BRSET TLIMIT, FLAG, FORCE

If temp out of range, force to -

LDX #58

JR TSTNEG

Convert 8 bit binary to 3 digit BCD.

CLRA

LUPDIS1 LSR DEC1

Shuffle bit order of digits to allow

for SCI lab first and 14489 msb first

incompatability.

LDA LUPDIS1

DEC2

BEQ

ADD #32

Add 32 to get degF.

STA NEWTMP

RTS

*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*
253 0000f98 ae08 LDX $8
254 0000f9a 4f CLRA
255 0000f9b 3459 LUPDIS2 LSR DEC0
256 0000f9d 49 ROLA
257 0000f9e 5a DECX
258 0000f9f 26fa BNE LUPDIS2
259 0000fa0 aa0f ORA #$0F
260 0000fa1 b75f STA DISP2
261 0000fa2 a631 LDA #$31
262 0000fa7 015502 BRCLR IMP,MODE,STDIS3
263 0000faa ae01 LDA #$F1
264 0000fac b760 STDIS3 STA DISP3
265 0000fba a6cb LDA #$CB
266 0000fb0 be57 LDX DEC2
267 0000fb2 2702 BEQ STDISC
268 0000fb4 a63b LDA #$3B
269 0000fb6 b761 STDISC STA DISPC
270 0000fbb 81 RTS
271 0000fbc ae6b FORCE LDA #$BB
272 0000fbd b75e, STA DISP1
273 0000fbf b75f STA DISP2
274 0000fc1 a631 LDA #$31
275 0000fc3 015502 BRCLR IMP,MODE,STDIS3
276 0000fc6 ae01 LDA #$F1
277 0000fc8 b760 STDIS3 STA DISP3
278 0000fca a6fb LDA #$FB
279 0000fcb b761 STA DISPC
280 0000fcd 81 RTS
281
282
283
284
285
286
287
288
289
290
291 0000fcb 0b132a TOVINT BRC LR TOF,TSR,NOOVF Check Tim overflow has really happened.
292 0000fcd 3c50 INC TICKS
293 0000fde a108 LDA TICKS
294 0000fdd 81
295 0000fda 3f50 CMP #8
296 0000feb 3c51 CLR TICKS
297 0000fed 1654 BSET SEC,FLAG
298 0000fe0 b651 LDA SECS
299 0000fe2 a13c CMP #$60
300 0000fe4 2514 BLO NOINC
301 0000fe6 3f51 CLR SECS
302 0000fe8 3c52 INC MINS
303 0000feda 3f51 CLR SECS
304 0000feb a652 LDA MINS
305 0000fed a13c CMP #$60
306 0000ff0 250a BLO NOINC
307 0000ff2 3f52 CLR MINS
308 0000ff4 b653 LDA HRS
309 0000ff6 a1ff CMP #$FF
310 0000ff8 2702 BEQ NOINC
311 0000ff8 3c53 INC HRS
312
313 0000ffa b619 NOINC LDA TIMLO Clear TOF flag.
314 0000fff 80 NOOVF RTI
315
43
ADCONV EQU *

BCLR TLIMIT,FLAG

CLR TEMP

BRSET 1,PORTC,CONT1 Check Thermistor selector switch.

LDA #$21

BRA SETAD

LDA #$20

STA ADSTCT Start first conversion.

LDX #'4 Init counter.

CLRA

BRLCR COCO,ADSTCT,ADLUP1 Wait for end of conversion.

ADD ADDATA

BCC DECCX Convert 4 times and accumulate to help eliminate noise.

INC TEMP

DECCX DECX

BNE ADLUP1

ROR TEMP

ROR TEMP

ROR TEMP

ROR TEMP

INIT TEMP

ADTAB,X and use them to access the look-up table

LDA ADTAB,X

CMP #$08 Check table entry limits.

BEQ TRANGE

BEQ TRANGE Store “base” value.

STA TEMP1

INCX

SUB ADTAB,X Get the diff between the base and next entry.

STA TEMP2

LDA TEMP

AND #$0F

Now get the lower 4 bits of the A/D result.

LDX TEMP2

MUL Multiply by the difference.

ROLA Divide answer by 16 and leave in TEMP2.

ROLA

ROLX

ROLA

ROLX

ROLA

ROLX

ROLA

ROLX

ROLA

ROLX

STX TEMP2

LDA TEMP1 Retrieve base value.

SUB TEMP2 subtract the difference value

STA NEWTMP and store answer in NEWTMP.

RTS

TRANGE BSET TLIMIT,FLAG

RTS

NEWTMP
CONBCD - Converts Binary in BIN0 to BCD in DECO-2

CONBCD EQU *

Clear BCD result bytes.

Put the next binary bit in carry.

Multiply current result by 2 and add in new bit at same time.

Now check the BCD bytes for overflow.

BCD number has overflowed so set flag and set upper digit to 9.

Any more bits to do?

Only update display registers if any of them have changed since the last time.

Clock idle low, edge in mid data, last clk.

131.072KHz baud with 4.19etc XTAL.

Transmit enabled.

Wait for preamble to finish.

Enable transmission to 14489.
443 000010a8 b660 LDA DISP3
444 000010aa b764 STA OLD03
445 000010ac b711 STA SCDA T Send first byte.
446 000010ae 0f10fd DWAIT1 BCLR TORE,SCSR, DWAIT1 Wait until it has been transferred
447 000010b1 b65f LDA DISP2 - then load second.
448 000010b3 b763 STA OLD02
449 000010b5 b711 STA SCDAT
450 000010b7 0f10fd DWAIT2 BCLR TORE,SCSR, DWAIT2
451 000010ba b66e LDA DISP1
452 000010bc b762 STA OLD01
453 000010be b711 STA SCDAT
454 000010c0 0d10fd DWAIT3 BCLR TC,SCSR, DWAIT3 Wait until 3rd byte has actually gone
455 000010c3 b600 LDA $00
456 000010c5 b70f STA SCCR2 Dissable SCI transmissions,
457 000010c7 1402 BSET 2,PORTC then disable 14489.
458 000010c9 81 RTS
459
460
461
462
463
464
465
466
467
468 000010ca DISCON EQU *
469 000010ca b661 LDA DISPC Only update config register if it has
470 000010cc b165 CMP OLD0C changed since last time.
471 000010ce 2601 BNE UPDCON
472 000010d0 81 RTS
473
474 000010d1 a601 UPDCON LDA $01
475 000010d3 b70e STA SCCR1 Clock idle low, edge in mid data, last clk.
476 000010d5 4a DECA
477 000010d6 b70d STA BAUD 131.072KHz baud with 4.19etc XTAL.
478 000010d8 a608 LDA $08
479 000010da b70f STA SCCR2 Transmit enabled.
480 000010dc 0d10fd PREAM1 BCLR TC,SCSR,PREAM1 Wait for preamble to finish.
481 000010df 1502 DOCONF BCLR 2,PORTC Enable transmission to 14489.
482 000010e1 b661 LDA DISPC
483 000010e3 b765 STA OLD0C
484 000010e5 b711 STA SCDAT
485 000010e7 0f10fd DWAIT4 BCLR TORE,SCSR, DWAIT4 Wait until config byte has transfered.
486 000010ea a600 LDA $00
487 000010ec b70f STA SCCR2 Now disable SCI transmission.
488 000010ee 0d10fd DWAIT5 BCLR TC,SCSR, DWAIT5 Wait until config byte has actually gone.
489 000010f1 1402 BSET 2,PORTC Disable 14489 & return.
490 000010f3 81 RTS
491
492
493
494
495
496
497
498
499
500 00001ff2 0f00 SCIINT FDB RESET
501 00001ff4 0f00 TOVFLOM FDB TOVFINT
502 00001ff6 0f00 TOCM FDB RESET
503 00001fff 0f00 TICAP FDB RESET
504 00001ffe 0f00 EXTINT FDB RESET
505 00001ffe 0f00 SOFTI FDB RESET
506 00001ffe 0f00 POR FDB RESET

************************************************************************
VECTOR ADDRESSES
************************************************************************
SECTION .VECT, ADDR=$1FF2

46
## Section synopsis

1 00000016 (22) .RAM
2 00000010 (16) .PAGE0
3 000001f4 (500) .USROM
4 0000000e (14) .VECT

## Symbol table

```
| Symbol   | Offset  |
|----------|--|---|
| .PAGE0   | 2 00000000 | ... |
| .RAM     | 1 00000000 | ... |
| .USROM   | 3 00000000 | ... |
| .VECT    | 4 00000000 | ... |
| .ADLUP1  | 3 0000100f | ... |
| .ADTAB   | 2 00000200 | ... |
| .BING    | 1 00000056 | ... |
| .CON1    | 3 00001008 | ... |
| .DECO    | 1 00000059 | ... |
| .DEC1    | 1 00000058 | ... |
| .DEC2    | 1 00000057 | ... |
| .DECCX   | 3 00001018 | ... |
| .DISP1   | 1 0000005e | ... |
| .DISP2   | 1 0000005f | ... |
| .DISP3   | 1 00000060 | ... |
| .DISPC   | 3 00000f5e | ... |
| .DOCONF  | 3 000010df | ... |
| .EXTINT  | 4 000010ae | ... |
| .FORCE   | 3 00000f83 | ... |
| .GODISP  | 3 00000f83 | ... |
| .GOMETR  | 3 00000f3a | ... |
| .GOMORE  | 1 0000005a | ... |
| .HRS     | 1 00000052 | ... |
| .INIRAM  | 3 00000f5e | ... |
```

## Symbol cross-reference

```
| Symbol   | Offset  |
|----------|--|---|
| .PAGE0   | *131 |
| .RAM     | *94  |
| .USROM   | *142 |
| .VECT    | *98  |
| .ADLUP1  | 334  |
| .ADTAB   | 333  |
| .BING    | 111  |
| .CON1    | 327  |
| .DECO    | 114  |
| .DEC1    | 113  |
| .DEC2    | 112  |
| .DECCX   | 336  |
| .DISP1   | 121  |
| .DISP2   | 122  |
| .DISP3   | 123  |
| .DISPC   | 124  |
| .DOCONF  | 481  |
| .DWAIT1  | 446  |
| .DWAIT2  | 450  |
| .DWAIT3  | 454  |
| .DWAIT4  | 485  |
| .DWAIT5  | 488  |
| .EXTINT  | 504  |
| .FORCE   | 237  |
| .GODISP  | 184  |
| .GOMETR  | 177  |
| .GOMORE  | 180  |
| .HRS     | 100  |
| .INIRAM  | 151  |
| .LUPBCD  | 392  |
| .LUPDIS1 | 242  |
| .LUPDIS2 | 255  |
| .MINS    | *99  |
| .MODE    | *108 |
```

47
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Cross-referenced Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULIP8</td>
<td>201 203 *207</td>
</tr>
<tr>
<td>NEWTMP</td>
<td>*116 179 197 226 326</td>
</tr>
<tr>
<td>NOIMP</td>
<td>173 *175</td>
</tr>
<tr>
<td>NOINC</td>
<td>295 301 306 310 *313</td>
</tr>
<tr>
<td>NONEG</td>
<td>198 *202</td>
</tr>
<tr>
<td>NONEG1</td>
<td>222 *224</td>
</tr>
<tr>
<td>NOOIF</td>
<td>291 *314</td>
</tr>
<tr>
<td>NOOVR</td>
<td>408 *412</td>
</tr>
<tr>
<td>OLDD1</td>
<td>*125 425 452</td>
</tr>
<tr>
<td>OLDD2</td>
<td>*126 428 448</td>
</tr>
<tr>
<td>OLDD3</td>
<td>*127 431 444</td>
</tr>
<tr>
<td>OLDDC</td>
<td>*128 150 470 483</td>
</tr>
<tr>
<td>POR</td>
<td>*506</td>
</tr>
<tr>
<td>PREAM</td>
<td>*441 441</td>
</tr>
<tr>
<td>PREAM1</td>
<td>*480 480</td>
</tr>
<tr>
<td>SCIINT</td>
<td>*500</td>
</tr>
<tr>
<td>SECS</td>
<td>*98 297 299 302</td>
</tr>
<tr>
<td>SETAD</td>
<td>329 *331</td>
</tr>
<tr>
<td>SOFTI</td>
<td>*505</td>
</tr>
<tr>
<td>STD1</td>
<td>249 250 *252</td>
</tr>
<tr>
<td>STDI3</td>
<td>276 *278</td>
</tr>
<tr>
<td>STDIS3</td>
<td>262 *264</td>
</tr>
<tr>
<td>STDISC</td>
<td>267 *269</td>
</tr>
<tr>
<td>TABL</td>
<td>*346</td>
</tr>
</tbody>
</table>

**Symbol cross-reference**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Cross-referenced Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMP</td>
<td>*117 326 337 340 342 344 346 359</td>
</tr>
<tr>
<td>TEMP1</td>
<td>*118 355 372</td>
</tr>
<tr>
<td>TEMP2</td>
<td>*119 358 361 371 373</td>
</tr>
<tr>
<td>TICAP</td>
<td>*503</td>
</tr>
<tr>
<td>TICKS</td>
<td>*97 292 293 296</td>
</tr>
<tr>
<td>TIMINT</td>
<td>*161</td>
</tr>
<tr>
<td>TOCMP</td>
<td>*502</td>
</tr>
<tr>
<td>TOVFELW</td>
<td>*501</td>
</tr>
<tr>
<td>TOVINT</td>
<td>*291 501</td>
</tr>
<tr>
<td>TRANGE</td>
<td>352 354 *376</td>
</tr>
<tr>
<td>TSTD1</td>
<td>398 *401</td>
</tr>
<tr>
<td>TSTD2</td>
<td>403 *406</td>
</tr>
<tr>
<td>TSTNEG</td>
<td>247 *250</td>
</tr>
<tr>
<td>UPDATE</td>
<td>426 429 432 *435</td>
</tr>
<tr>
<td>UPDCON</td>
<td>471 *474</td>
</tr>
</tbody>
</table>
128K byte addressing with the M68HC11

By Ross Mitchell
MCU Applications Engineering
Motorola Ltd., East Kilbride, Scotland

OVERVIEW

The maximum direct addressing capability of the M68HC11 device is 64K bytes, but this can be insufficient for some applications. This application note describes two methods of memory paging that allow the MCU to fully address a single 1 megabit EPROM (128K bytes) by manipulation of the address lines.

The two methods illustrate the concept of paging and the inherent compromises. The technique may be expanded to allow addressing of several EPROM, RAM or EEPROM memories or several smaller memories by using both address lines and chip enables.

PAGING SCHEME

The M68HC11 8-bit MCU is capable of addressing up to 64K bytes of contiguous address space. Addressing greater than 64K bytes requires that a section of the memory be replaced with another block of memory at the same address range. This technique of swapping memory is known as paging and is simply a method of overlaying blocks of data over each other such that only one of the blocks or pages is visible to the CPU at a given time.

In a system requiring more than 64K bytes of user code and tables, it is possible to use the port lines to extend the memory addressing range of the M68HC11 device. This has certain restrictions but these can be minimised by careful consideration of the user code implementation.

There are two basic configurations; method A uses only software plus a single port line to control the high address bit A16; method B is a combination of a small amount of hardware and software controlling the top 3 address bits A14, A15 and A16.

In the examples below, the MC68HC11G5 device is used to demonstrate the paging techniques since this device has a non-multiplexed data and address bus; any M68HC11 device may be used in a similar way.

Method A has the advantage of no additional hardware and very few limitations in the software. The user code main loop can be up to 64K bytes long and remain in the same page but this is at the expense of longer interrupt latency. The vector table and a small amount of code must be present in both pages of memory to allow correct swapping of the pages.

Method B has the advantage of not affecting the interrupt latency and has just one copy of the vector table. The maximum length of the user code main loop in this example is 48K bytes with a further 5 paged areas of 16K bytes for subroutines and tables.
SOFTWARE TECHNIQUE

Address A16 of the EPROM is directly controlled by port D(5) of the M68HC11 as shown in figure 1. This port is automatically configured to be in the input state following reset. It is vital that the state of the port line controlling address A16 is known following reset and so there is a 10KΩ pull-up resistor on this port line to force the A16 address bit to a logic high state following reset. This port bit is then made an output during the set-up code execution but care must be taken in ensuring that the data register is written to a logic one before the data direction register is written with a one to make the port line output a high state.

This port bit allows the M68HC11 to access the 128K byte EPROM as two memories of 64K bytes each which are paged by changing the state of the address A16 line on the EPROM. It important to make sure that the port timing enables the port line to change state at least the setup and hold time before the address strobe (E clock rising edge on the MC68HC11G5), otherwise there could be problems with address timing.

Figure 2 shows a schematic representation of the paging technique for this method where there are two separate 64K byte pages of memory which may only be addressed individually.

This paging scheme means that code cannot directly jump from one 64K page to another without running some common area of code during the page switch. This may be accomplished in 2 basic ways. The user code could build a routine in RAM (which is common to both pages since it is internal and therefore unaffected by the port D(5) line) or have the same location in both pages devoted to a page change routine. The example software listing in appendix A uses the latter approach.

Interrupt routines

The change of page routine stores the current page before setting or clearing the port D(5) line and then has a jump command which must be at exactly the same address in both pages of memory. This is because the setting or clearing of the port D(5) line will immediately change the page of memory but the program counter will increment normally. Thus a change from page 0 to page 1 will result in the BSET PORTD command from page 0 followed by the JMP 0,X instruction from page 1 (the new page). To enable a jump to work, the X index register has been loaded with the address of the routine to be run in the new page. Figure 3 shows the execution of code to perform a change of page from page 1 to page 0.

Returning from the interrupt routine requires the RTI command to be replaced with a return from interrupt routine that checks the RAM location containing the memory page number prior to the interrupt routine execution. The routine then either performs an RTI command immediately if it is to remain in the same page or otherwise changes the state of the port D(5) line and then performs an RTI command in the correct page. Note that as with the JMP 0,X command, the RTI must be at the same address in both pages. It is important that the 1-bit in the CCR (interrupt inhibit) is set during this time for the example code to run correctly, otherwise the return page may be altered. This limitation can be overcome by using the stack to maintain a copy of the last page prior to the current interrupt.

The latency for an interrupt routine in a different page from the currently running user code is increased by 21 cycles on entering the interrupt routine and 18 cycles on leaving the interrupt routine. Any interrupt code that could not tolerate any such latency could be repeated in both pages of memory.

Other routines

Jumping from one page to another may be done at any time by using the same change of page routine but there is no need to store the current page in RAM and so these two lines of code become redundant. In the example, the change page routine could be started at the BCLR or BSET command and save 4 cycles. This would therefore reduce the page change delay to 17 cycles. Note that it is not possible to perform a JSR command to move into the other page with the method shown in the example since the RTS would not return to the original page, however, a modification to the return from interrupt routine would allow an equivalent function for a return from subroutine. In this case the stack should be used to maintain the correct return page or the 1-bit in the CCR should be set to prevent interrupts.

Important conditions

The state of the port line controlling address A16 after reset is very important. In the example, port D(5) is used which is an input after reset and has a pull-up resistor to force a logic high on A16. If an output port line was used then it could be reset such that A16 is a logic zero (no pull-up resistor required) which has an important consequence. The initialisation routine which sets up the ports must be in the default page dictated by the state of address A16 following reset otherwise the user code may not be able to correctly configure the ports and hence be unable to manipulate address A16. Similarly, a bidirectional port line could have a pull-down resistor to determine the address A16 line after reset with the same implications.

The assembler generates two blocks of code with identical address ranges used by the user code. This could not be programmed directly into an EPROM since the second page would simply attempt to overwrite the first page. The code must therefore be split into two blocks and programmed into the correct half of the EPROM. Some linkers may be capable of performing this function automatically. Figure 2 illustrates the expansion of the pages into the 128K byte EPROM memory.

The RAM and registers, and internal EEPROM if available and enabled, will all appear in the memory map in preference to external memory so care must be taken to avoid these addresses or move the RAM or registers away to different addresses by writing to the INIT register.
Figure 1. Software Paging Schematic Diagram

Figure 2. Software Paging Representation
Figure 3. Flow of program changing from Page 1 to Page 0

1 - Jump to change page routine
2 - Page changes to page 0
3 - Jump to address in X register (in page 0)
Figure 4. Hardware and Software Paging Schematic Diagram
METHOD B – COMBINED HARDWARE AND SOFTWARE TECHNIQUE

The basic approach to this method is the same as above except that hardware replaces some of the software. A port line together with M68HC11 addresses A14 and A15 are NOR’d to control the address A16 line of the EPROM. This signal is also used to select between the port line and address line for A14 and A15 (see figure 4). The hardware between the port lines controlling the A14 and A15 addresses enables 64K bytes of user code to be addressed at all times with 48K bytes common to all the pages and then selecting one of five 16K bytes pages of EPROM memory.

In the example, port D(3) and address A14 are connected to the input of a 2 channel multiplexer such that port D(5), address A14 and address A15 control which of these two signals reaches the A14 pin of the EPROM. If addresses A14 or A15 are logic 1, the NOR gate outputs a logic 0 state, ensuring the A16 pin of the EPROM is a logic 0. In this case address A14 controls the A14 pin of the EPROM and similarly A15 and port D(4) are selected such that address A15 controls the A15 pin of the EPROM. Thus the main 48K byte portion of the EPROM memory may be addressed at all times at addresses $4000$ up to $FFFF$. With Port D(5) and address A14 and A15 all at logic 0 (address range $0000$ to $3FFF$), the port lines Port D(3) and Port D(4) are selected in place of address lines A14 and A15. Page 0 is always selected whenever Port D(5) is a logic 1. This makes it possible to have one of the five pages of 16 K bytes paged into the 64K addressing range of the HC11 while always maintaining the main 48K bytes of user code in the memory map.

There are few restrictions on the user code since the hardware provides the switching logic. Code can be made to run from one paged area to another by jumping to an intermediate routine in the main page. Port D is configured to be in the input state following reset which results in the main page plus page 0 of the paged memory in the 64K byte address map since the port D lines each have a pull-up resistor to maintain a logic high state after reset. A simple change memory map routine can then bring in the desired page at any time. Appendix B shows the assembly code for a program that toggles different port pins in each of the 5 pages controlled from a main routine in the main page. Figure 5 shows the 5 overlaid pages expanded to a 128K map with the flow of the program demonstrating a change from page 0 to page 1 by running the change page subroutine shown in bold type.

Implementation in ‘C’ language

The demonstration code was originally written in assembly language but it may also be implemented in ‘C’ as shown in appendix C. The change of page routines were written in ‘C’ with the first part an example of using in-line code and the second part calling a function. The short example shows the assembly code on the left, generated by the ‘C’ code on the right. This is very similar to the assembly code example in appendix B and so it is possible to extend the memory addressing beyond 64K bytes with the ‘C’ language just as with assembly language.

Interrupt conditions

The interrupt routines have normal latency when they reside in the main 48K bytes page since this is always visible to the CPU. The 25 cycle delay for changing pages may cause problems for interrupt routines in a paged area of memory.

Important conditions

There are few special conditions for this method. The vector must point to the main page of memory where the page changing routine must also reside. Routines in a paged area can only move to another page via the main 48K page unless the technique in method A is utilised (i.e. page change routine duplicated at identical addresses in both pages).

As with method A, the RAM and registers, and internal EEPROM if available and enabled, will all appear in the memory map in preference to external memory so care must be taken to avoid these addresses or move the RAM or registers away to different addresses.

The assembler generates 5 blocks of code with identical address ranges used by the user code plus the main 48K byte section. This could not be programmed directly into an EPROM since the second and subsequent pages would simply attempt to overwrite the first page. The code must therefore be split into blocks and programmed into the correct part of the EPROM. Some linkers may be capable of performing this function automatically.

Figure 6 illustrates the expansion of the pages into a single 128K byte EPROM memory.

Customisation

Clearly the size of the paged areas may be made to suit the application with for example a 32K byte main page and three 32K bytes of paged memory simply by not implementing control over the A14 address of the EPROM and not including Port D(3) control. Similarly by adding another port line to control address A13, the main program can be 56K bytes with 9 pages of 8K bytes each.
TOGGLE PORT A-3
AND RETURN TO MAIN PROGRAM

SET UP PORT D FOR PAGE CONTROL
JSR TO PAGE 0 CHANGE SUBROUTINE
JSR TO PAGE 0

JSR TO PAGE 1 CHANGE SUBROUTINE
JSR TO PAGE 1

JSR TO PAGE 2 CHANGE SUBROUTINE
JSR TO PAGE 2
e tc

CHANGE TO PAGE 0 AND RETURN
CHANGE TO PAGE 1 AND RETURN
CHANGE TO PAGE 2 AND RETURN
e tc

VECTORS

TOGGLE PORT A-4
AND RETURN TO MAIN PROGRAM

TOGGLE PORT A-5
AND RETURN TO MAIN PROGRAM

TOGGLE PORT A-6
AND RETURN TO MAIN PROGRAM

TOGGLE PORT A-7
AND RETURN TO MAIN PROGRAM

1 - Return from page 0
2 - Jump to page 1 routine
3 - Return from page 1 to main page

Figure 5. Illustration of changing from Page 0 to Page 1
Figure 6. Hardware and software paging representation
IN GENERAL

In both methods, the registers may be moved to more appropriate addresses. If the usage of RAM is not critical the registers may be moved to address $0000 by writing $00 to the INIT register immediately after reset. For the MC68HC11G5 this means losing 128 bytes of RAM but results in a clean memory map above $1FF. In the examples, the registers and RAM remain at the default addresses and so care must be taken not to have user code from address $0000 to $01FF and $1000 to $107F for the MC68HC11G5. Note that the MC68HC11E9 and MC68HC11A8 have slightly different RAM and register address ranges plus the internal EEPROM which should be disabled if not used.

Figure 7 demonstrates the differences between the paging techniques by showing the overlap of the pages. The number and size of the pages can easily be modified by small changes to the page change routines and hardware.

Beyond 128K bytes

Both techniques may be scaled up with several port lines controlling address lines beyond address A15 with the addition of further change page routines and enhancing the return from interrupt routine to allow a return to a specific page in method A or the addition of further multiplexing logic in method B.

IN CONCLUSION

The two methods described in detail are the basis for many other ways of controlling paging on a single large EPROM memory device or several smaller EPROMs. It is a simple matter to scale up or modify the techniques to suit a particular application or EPROM. The software approach is the cheapest and allows for a main program of up to the full size of the EPROM while the combined hardware and software approach has a maximum main program size of 48K bytes (in this example) and no additional interrupt latency.
APPENDIX A - SOFTWARE PAGING SCHEME

**** EXTENDA.ASC *****************************************
* TESTS EXTENDED MEMORY CONTROL
* For a single 1M bit (128K byte) EPROM split into 2 x 64K byte pages.
* A16 is connected to Port D(5) which then selects which half of
* the EPROM is being accessed. PD5 = 1 after reset since it is in
* the input state with a pull-up resistor to Vdd.
* This code is written for the 68HC11G5 MCU but can be easily modified
* to run on any 68HC11 device. The 68HC11G5 has a non-multiplexed
* address and data bus in expanded mode.

**********************************************************************
** PORTA EOU S00
** DDRA EQU $00
** PORTB EQU $04
** PORTC EQU $06
** DDRC EQU $07
** PORTD EQU $08
** DDRD EQU $09
** TMSK2 EQU $24
** TFLG2 EQU $25
** RTII EQU $40
** RTIF EQU $40
** PACTL EQU $26
** DDRA7 EQU $80
** REGS EQU $1000
**
** RAM definitions (from $0000 to $01FF)
**
** ORG $0000
** PAGE RMB 1 page number prior to interrupt
** TIME RMB 2 counter value for real time interrupt routine
** NPAGE EQU $20 PORT D-5 page control line
** ROMBASE EQU $0200 Avoid RAM (from $0 to $1FF)
** CHANGE EQU $F800
** VECTORS EQU $FFCC
**
** START OF MAIN PROGRAM
**********************************************************************
** ORG ROMBASE
** Redirect reset vector to page 1

58
63 00000200 ce0200 RESETO LD X #RESET
64 00000203 7ef800 JMP CH GPAGEO
65 66 67 68 69 70
71 00000206 181c0010 LOOP PO BSET PORTA,Y,#$10 Toggle bit 4
72 0000020a 181d0010 BCLR PORTA,Y,#$10
73 0000020e ce0216 LD X #LOO P11 get return address in page 1
74 00000211 7ef800 JMP CH GPAGEO jump to change page routine
75 76 77 78 79
80 81 00000214 181e254001 LOOPPO BSET PORTA,Y,#$10 Toggle bit 4
82 00000219 3b RTI return if not correct interrupt source
83 * This is an RTI because interrupt vector only points here when in page 1
84 85 86 0000021a RTISRV BSET TFLG2,Y,#$10,RTISERV
87 0000021c 8640 LDAA #$01000000 page 0 interrupt starts here
88 0000021f 9602 STAA TFLG2,Y clear RTI flag
89 00000221 8c INCA INcrement counter
90 00000222 b71004 STAA PORTB+REGS store time in port B
91 00000225 de01 LD X TIME
92 00000227 08 STAA INCREMENT
93 00000228 df01 STX TIME and copy back into RAM
94 0000022a 7ef80a JMP RETRTIO jump to RTI routine
95 96
97 98 99
100 101 102
104
105
106
107
108
109
110
111
112
113
114
115 0000f800 ORG CHANGE Address for this routine is fixed cycles
116 0000f800 8600 LDA A #0 2 set current page number = 0
117 0000f802 9700 STA A PAGE 2 store page page number
118 0000f804 181c0820 BSET PORTD,Y,#$PAGE 8 change page by setting PD-5
119 0000f808 6e00 JMP 0,X 3 This code is the same in both pages
120 121 122
126
127
128

59
* cycles
129 0000f80a RETRTIO 2 get page the interrupt occured in
130 0000f80a 9600 LOOA PAGE 2 is it page 1
131 0000f80c 8101 CMPA #1 3 if yes then change page
132 0000f80e 2701 BEQ RTIPAGEO 12 otherwise, return from interrupt
133 0000f810 3b RTI
134 0000f811 RTIPAGEO
135 0000f811 181c0820 BSET PORTD, Y, #PAGE 8 change page and return from interrupt
136 0000f815 3b RTI 12 This codes is the same in both pages
137
138
139
140
141
142
143
144
145 0000ffcc 0200 ORG VECTORS
146 0000ffce 0200 FDB RESETO EVENT 2
147 0000ffd0 0200 FDB RESETO EVENT 1
148 0000ffd2 0200 FDB RESETO TIMER OVERFLOW 2
149 0000ffd4 0200 FDB RESETO INPUT CAPTURE 6 / OUTPUT COMPARE 7
150 0000ffd6 0200 FDB RESETO INPUT CAPTURE 5 / OUTPUT COMPARE 6
151 0000ffd8 0200 FDB RESETO SCI
152 0000ffda 0200 FDB RESETO SPI
153 0000ffdc 0200 FDB RESETO PULSE ACC INPUT
154 0000ffde 0200 FDB RESETO PULSE ACC OVERFLOW
155 0000ffe0 0200 FDB RESETO TIMER OVERFLOW 1
156 0000ffe2 0200 FDB RESETO INPUT CAPTURE 4 / OUTPUT COMPARE 5
157 0000ffe4 0200 FDB RESETO OUTPUT COMPARE 4
158 0000ffe6 0200 FDB RESETO OUTPUT COMPARE 3
159 0000ffe8 0200 FDB RESETO OUTPUT COMPARE 2
160 0000ffe0 0200 FDB RESETO OUTPUT COMPARE 1
161 0000ffe2 0200 FDB RESETO INPUT CAPTURE 3
162 0000ffe4 0200 FDB RESETO INPUT CAPTURE 2
163 0000ffe6 0200 FDB RESETO INPUT CAPTURE 1
164 0000ffe8 0200 FDB RTISRV REAL TIME INTERRUPT
165 0000fffa 0200 FDB RESETO IRQ
166 0000fffc 0200 FDB RESETO XIRQ
167 0000fffd 0200 FDB RESETO SWI
168 0000ffe8 0200 FDB RESETO ILLEGAL OPCODE
169 0000ffee 0200 FDB RESETO COP
170 0000ffee 0200 FDB RESETO CLOCK MONITOR
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186 00000200 8e01ff ORG ROMBASE
187 00000200 bd021b RESET LDS #$01FF initialise RTI interrupt and DDRs
188 00000206 86ff JSR SETUP
189 00000208 181c0008 LOOP BSET PORTA, Y, #$08 Toggle bit 3
190 0000020c 181d0008 BCLR PORTA, Y, #$08
191 00000210 ce0206 LDX #LOOPPO set up jump to other page
192 00000213 7ef800 JMP CHGPAGE1 go to other page
193 00000216 LOOP1
194 00000216 4a DECA return point from other page
195 00000217 26ef BRA LOOP toggle port A
196 00000219 20eb BRA LOOP1 start loop again
INITIALISATION ROUTINE

SETUP SEI

LDY #$1000  Register address offset

LDAA #$FF  make port A all outputs

STAA DDRA+REGS make sure port D-5 is written a 1

STAA DDRD+REGS make sure port 0-5 is written a 1

STAA PORTD+REGS make sure port 0-5 is written a 1

LDAA #$01000000

STAA TFLG2+REGS clear RTI flag

STAA TMSK2+REGS enable RTI interrupt

CLI

RTS

Redirect to the Real time interrupt service routine

Page 1 routine for service routines located in page 0

******************************************

INTRTI BRSET TFLG2,Y,1RTIF,G00DINT

RTI return if not correct interrupt source

This is an RTI because interrupt vector only points here when in page 1

GOODINT cycles

LDX #RTISERV 3 get the interrupt entry point in page 0

JMP CHGPAGE1 3 jump to change page routine

CHANGE PAGE ROUTINE

This code must be executed with the I-bit set to prevent interrupts during the change if it is a jump for an interrupt routine.

Otherwise PAGE could be updated and then another interrupt could occur before the PAGE was changed causing the first interrupt routine to return to the wrong page.

The PAGE variable is not required for a normal jump and so it does not require the I-bit to be set (only the BCLR is important).

This code is repeated for the same position in both pages

jump routine

ORG CHANGE Address for this routine is fixed

cycles

CHGPAGE1

LDAA #$1 2 set current page number = 1

STAA PAGE 2 store page number

BCLR PORTD,Y,#PAGE 8 change page by clearing PD-5

JMP 0,X 3 This code is the same in both pages

return from interrupt routine running in page 0

check if interrupt occurred while code was running in page 1 and return to page 0 before the RTI command is performed

***********************************************************************
260 cycles
261 RETRTI1
262 LDA PAGE  2 get page the interrupt occurred in
263 CMPA #0  2 is it page 0
264 BEQ RTIPAGE1  3 if yes then change page
265 RTI  12 otherwise, return from interrupt
266 RTIPAGE1
267 BCLR PORTD,Y,#NPAGE  8 change page and return from interrupt
268 This code is the same in both pages
269
270
271
272
273
274
275
276 ORG VECTORS
277 FDB RESET EVENT 2
278 FDB RESET EVENT 1
279 FDB RESET TIMER OVERFLOW 2
280 FDB RESET INPUT CAPTURE 6 / OUTPUT COMPARE 7
281 FDB RESET INPUT CAPTURE 5 / OUTPUT COMPARE 6
282 FDB RESET SCI
283 FDB RESET PULSE ACC INPUT
284 FDB RESET PULSE ACC OVERFLOW
285 FDB RESET TIMER OVERFLOW 1
286 FDB RESET INPUT CAPTURE 4 / OUTPUT COMPARE 5
287 FDB RESET OUTPUT COMPARE 4
288 FDB RESET OUTPUT COMPARE 3
289 FDB RESET OUTPUT COMPARE 2
290 FDB RESET OUTPUT COMPARE 1
291 FDB RESET INPUT CAPTURE 3
292 FDB RESET INPUT CAPTURE 2
293 FDB RESET INPUT CAPTURE 1
294 FDB INTRTI REAL TIME INTERRUPT
295 FDB RESET IRQ
296 FDB RESET XIRQ
297 FDB RESET SWI
298 FDB RESET ILLEGAL OPCODE
299 FDB RESET COP
300 FDB RESET CLOCK MONITOR
301 FDB RESET RESET
302 END

62
APPENDIX B - HARDWARE AND SOFTWARE PAGING SCHEME

******* EXTENDB.ASC ****************************

* TESTS EXTENDED MEMORY CONTROL

* for a single 1M bit (128K byte) EEPROM split into 48KB + 5 x 16KB
* $4000 - $FFFF 48K COMMON PAGE
* $0200 - $3FFF 16K PAGES 0,1,2,3,4

* A multiplexer is used to switch between address and port D lines
* controlled by PD5 and A16 is controlled by /(PD5+A14+A15)
* This ensures that Address A16 is a logic 1 whenever A14 or A15 are
* high and that all three lines must be low for the paged memory between
* addresses $00000 and $0FFFF.

<table>
<thead>
<tr>
<th>SOURCE CODE ADDRESS</th>
<th>EPROM ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0000</td>
<td>PAGE 0</td>
</tr>
<tr>
<td>4000</td>
<td>PAGE 0</td>
</tr>
<tr>
<td>0000</td>
<td>MAIN PAGE</td>
</tr>
<tr>
<td>0000</td>
<td>PAGE 0</td>
</tr>
<tr>
<td>10000</td>
<td>PAGE 1</td>
</tr>
<tr>
<td>0000</td>
<td>PAGE 1</td>
</tr>
<tr>
<td>14000</td>
<td>PAGE 2</td>
</tr>
<tr>
<td>0000</td>
<td>PAGE 2</td>
</tr>
<tr>
<td>18000</td>
<td>PAGE 3</td>
</tr>
<tr>
<td>0000</td>
<td>PAGE 3</td>
</tr>
<tr>
<td>1C000</td>
<td>PAGE 4</td>
</tr>
<tr>
<td>0000</td>
<td>PAGE 4</td>
</tr>
<tr>
<td>3FFF</td>
<td>1FFFF</td>
</tr>
</tbody>
</table>

(Continued overleaf)
PD3, PD4 AND PD5 = 1 AFTER RESET
SINCE PULL-UP RESISTORS FORCE HIGH STATE WITH PORT D AS INPUTS
WHICH DEFAULTS TO MAIN PROGRAM PLUS PAGE 0
**** ****************************************************.~***.P~***************

RAM definitions

**************************************************

ORG $0000

TIME RMB 2

Real time interrupt routine counter

*R

ROMBASE0 EQU $0200

Avoid RAM (from $0C to $1FF)

ROMBASE1 EQU $4000

VECTORS EQU $FFCC

PAGE 0 = $00000 - $03FFF

Page 0

PAGE 1 = $10000 - $13FFF

Page 1

PAGE 2 = $14000 - $17FFF

Page 2

PAGE 3 = $18000 - $1BFFF

Page 3

PAGE 4 = $1C000 - $1FFF

Page 4

Page n is added to 0xx000000 to give the state of port

START EQU $00

D(3), D(4) and D(5).

START EQU $00

PAGE0 EQU $20

PAGE1 EQU $00

PAGE2 EQU $08

PAGE3 EQU $10

PAGE4 EQU $18

65
* START OF MAIN PROGRAM *

** ORG ROMBASE0 **

00000200 181C0008 LOOP0 BSET PORTA,Y,$508

00000204 181D0008 BCLR PORTA,Y,$508

00000208 7E4014 JMP MAIN0 return to main page

* MAIN ROUTINE NOT UNDER INTERRUPT CONTROL *

** ORG ROMBASE1 **

00000400 8E01FF RESET LDS #$01FF initialise RTI interrupt and DDRs

00000403 BD402E JSR SETUP

00000406 181C0840 LOOP BSET PORTD,Y,$40 main routine toggles port D-2

0000040A 181D0840 BCLR PORTD,Y,$40

0000040E BD4062 JMP LOOP0 Toggle Port A-3

00000411 7E0200 JMP LOOP0 Toggle Port A-3

00000414 BD406D MAIN0 JSR CHGPAGE0 select page 0

00000417 BD0200 JSR LOOP1 Toggle Port A-4

0000041A BD4078 JSR CHGPAGE2 select page 2

0000041D BD0200 JSR LOOP2 Toggle Port A-5

00000420 BD4083 JSR CHGPAGE3 select page 3

00000423 7E0200 JMP LOOP3 Toggle Port A-6

00000426 BD408E MAIN3 JSR CHGPAGE4 select page 4

00000429 7E0200 JMP LOOP4 Toggle Port A-7

0000042C 20D8 MAIN4 BRA LOOP start loop again

* INITIALISATION ROUTINE *

** ORG ROMBASE1 **

0000042E OF SETUP SEI

0000042F 86FF LOAA #$01000 Register address offset

00000433 86FF LDAA #$FF make port A all outputs (68HC11G5)

00000435 B71001 STA PORTA+REGS

00000438 B71009 STA PORTD+REGS make port D all outputs

0000043B 7F0000 CLR TIME clear the RTI flag

0000043E 7F0001 CLR TIME+1 enable RTI interrupt

00000441 4F CLRA

00000442 B71000 STA PORTA+REGS

00000445 B71009 STA PORTD+REGS

00000448 8640 LDAA #$01000000

0000044A B71025 STA TFLAG2+REGS

0000044D B71024 STA TMSK2+REGS

00000450 0E CLI

00000451 39 RTS
Real time interrupt service routine

Real time interrupt service routine

RTISR V LDA A
STAA
STAA
LDAA
STAA
LOX
INX
STX
RTI

清RTI 标志

store counter in port B

get time counter

increment counter

save counter value in RAM

Return from interrupt

* CHANGE PAGE

acc B (bits 3-5) contains the 1's complement of new page number address

* SOURCE CODE EPROM

ADDRESS ADDRESS

0000 +----------------------+

| PAGE 0 |

| 4000 + - - - - - - - - + 04000 |

| PAGE 1 |

| 0000 +----------------------+

| PAGE 2 |

| 0000 +----------------------+

| PAGE 3 |

| 0000 +----------------------+

| PAGE 4 |

| 3FFF +----------------------+

PAGE 0 = $00000 - $03FF (A16=0,A15=0,A14=0) \(\rightarrow\) PAGE0=$00100000

MAIN = $04000 - $0FFFF (A16=0) \(\rightarrow\) START=$001XX000

PAGE 1 = $10000 - $13FFF (A16=1,A15=0,A14=0) \(\rightarrow\) PAGE1=$00000000

PAGE 2 = $14000 - $17FFF (A16=1,A15=0,A14=1) \(\rightarrow\) PAGE2=$00001000

PAGE 3 = $18000 - $1BFFF (A16=1,A15=1,A14=0) \(\rightarrow\) PAGE3=$00001000

PAGE 4 = $1C000 - $1FFFF (A16=1,A15=1,A14=1) \(\rightarrow\) PAGE4=$00011000

*
* CHGPAGE0
  LDAA PORTD+REGS
  ANDA #110000111
  ADDA #PAGE0
  RTS
  (only bits 3, 4 and 5 are changed)

* CHGPAGE1
  LDAA PORTD+REGS
  ANDA #110000111
  ADDA #PAGE1
  RTS
  (only bits 3, 4 and 5 are changed)

* CHGPAGE2
  LDAA PORTD+REGS
  ANDA #110000111
  ADDA #PAGE2
  RTS
  (only bits 3, 4 and 5 are changed)

* CHGPAGE3
  LDAA PORTD+REGS
  ANDA #110000111
  ADDA #PAGE3
  RTS
  (only bits 3, 4 and 5 are changed)

* CHGPAGE4
  LDAA PORTD+REGS
  ANDA #110000111
  ADDA #PAGE4
  RTS
  (only bits 3, 4 and 5 are changed)

* VECTORS

* ORG VECTORS
  EVENT 2
  EVENT 1
  TIMER OVERFLOW 2
  INPUT CAPTURE 6 / OUTPUT COMPARE 7
  INPUT CAPTURE 5 / OUTPUT COMPARE 6
  SCI
  SPI
  PULSE ACC INPUT
  PULSE ACC OVERFLOW
  TIMER OVERFLOW 1
  INPUT CAPTURE 4 / OUTPUT COMPARE 5
  OUTPUT COMPARE 4
  OUTPUT COMPARE 3
  OUTPUT COMPARE 2
**Page 1 (2nd half of EPROM)**

- **284 0000ffe8 4000** FDB RESET OUTPUT COMPARE 1
- **285 0000ffe9 4000** FDB RESET INPUT CAPTURE 3
- **286 0000ffeb 4000** FDB RESET INPUT CAPTURE 2
- **287 0000ffed 4000** FDB RESET INPUT CAPTURE 1
- **288 0000fff0 4052** FDB RTISR V REAL TIME INTERRUPT
- **289 0000fff2 4000** FDB RESET IRQ
- **290 0000fff4 4000** FDB RESET XIRQ
- **291 0000fff6 4000** FDB RESET SWI
- **292 0000fff8 4000** FDB RESET ILLEGAL OPCODE
- **293 0000fffc 4000** FDB RESET COP
- **294 0000fffe 4000** FDB RESET CLOCK MONITOR
- **295 0000fff0 4000** FDB RESET RESET

- **300** *page 2 (2nd half of EPROM)*
- **301** *
- **302** *
- **303** *

- **304** org ROMBASE0
- **305 00000020 181c0010** LOOPP1 BSET PORTA,Y,'$10 Toggle Port A-4
- **306 00000024 181d0010** BCLR PORTA,Y,'$10

- **309** *
- **310** *
- **311** *
- **312** *

- **313** *

- **314** org ROMBASE0
- **315 00000020 181c0020** LOOPP2 BSET PORTA,Y,'$20 Toggle Port A-5
- **316 00000024 181d0020** BCLR PORTA,Y,'$20

- **319** *
- **320** *
- **321** *
- **322** *

- **323** *

- **324** org ROMBASE0
- **325 00000020 181c0040** LOOPP3 BSET PORTA,Y,'$40 Toggle Port A-6
- **326 00000024 181d0040** BCLR PORTA,Y,'$40

- **329** *
- **330** *
- **331** *
- **332** *

- **333** *

- **334** org ROMBASE0
- **335 00000020 181c0080** LOOPP4 BSET PORTA,Y,'$80 Toggle Port A-7
- **336 00000024 181d0080** BCLR PORTA,Y,'$80

- **337 00000028 7e402c** JMP MAIN4 return to main page
- **338** *

- **339** *

**END**
APPENDIX C – ‘C’ LANGUAGE ROUTINES FOR METHOD B

* /* CHGPAGE.C */
* C coded extended memory control for 68HC11 */
* */
**************************************************************************************************************/
/* HC11 structure - I/O registers for MC68HC11 */

struct HC11IO {
    unsigned char PORTA; /* Port A - 3 input only, 5 output only */
    unsigned char Reserved; /* Motorola's unknown register */
    unsigned char PIOC; /* Parallel I/O control */
    unsigned char PORTC; /* Port C */
    unsigned char PORTB; /* Port B - Output only */
    unsigned char PORTCL; /* Alternate port C latch */
    unsigned char Reserved1; /* Motorola's unknown register 2 */
    unsigned char DDRC; /* Data direction for port C */
    unsigned char PORTD; /* Port D */
    unsigned char DDRD; /* Data direction for port D */
    unsigned char PORTE; /* Port E */
};
/* End of structure HC11IO */
*************************************************************************************************************/

#define regbase (*(struct HC11IO *) 0x1000)

typedef unsigned char byte;

/* Some arbitrary user defined values */
#define page0 0x20
#define page1 0x00
#define page2 0x08
#define pagemask 0xc7

/* Macro to generate in line code */
#define chgpage(a) regbase.PORTO = (regbase.PORTO & pagemask) + a

/* Function prototype */
void func_chgpage(byte p);

/* Externally defined functions in separate pages */
extern void func_in_page0(); /* Dummy function in page 0 */
extern void func_in_page2(); /* Dummy function in page 2 */
main() {
    chgpage(page2);
    /* Change page using inline code */

    func_in_page2();
    /* Call function in page 2 */

    func_chgpage(page0);
    /* Change page using function call */

    func_in_page0();
    /* Call function in page 0 */
}

void func_chgpage(p)
byte p;
{
    chgpage(p);
}

import func_in_page2
import func_in_page0
end
TV on-screen display using the MC68HC05T1

By Peter Topping
Motorola Ltd., East Kilbride, Scotland

INTRODUCTION

The "T" members of the MC68HC05 family of MCUs provide a convenient and cost-effective method of adding on-screen display (OSD) to TVs and VCRs. As well as the OSD capability, they include 8 Kbytes of ROM (adequate for Teletext, frequency-synthesis, stereo and OSD), 320 bytes of RAM, a 16-bit timer and 8 pulse-width-modulated D/A converters. The MC68HC05T7/8 also includes IIC hardware and, by using a 56/64-pin package, 4 ports of I/O independent of the OSD, serial and D/A outputs. It is thus suitable for large full-feature chassis. The MC68HC05T1 is in the middle of the price/performance range and includes most of the features of the MC68HC05T8 but in a 40-pin package. This is achieved by sharing I/O with the other pin functions (SPI, OSD, D/A). Even if all these features are used there is sufficient I/O for most applications. The low-cost MC68HC05T4 has 5 Kbytes of ROM and 96 bytes of RAM making it suitable for simpler (mono, non-Teletext) applications.

68HC05T1 OSD FEATURES

- Programmable display of 10 rows of 18 characters
- 24 byte (18 data + 6 control) single row architecture
- Settable in software to any one of four standards
- Zero inter-row and inter-column spacing
- 64 user-defined mask-programmable 8 x 13 characters
- Programmable horizontal position

- Character colour selectable from 4 colours/row
- Software programmable (start, stop and colour) window
- 4 character sizes (normal, double height and/or width)
- Half-dot character rounding
- Selectable half-dot black outline

OSD CHARACTERISTICS

The HC05TX series have an OSD capability of 10 rows of 18 characters. Each row can contain characters of four colours selected from the eight available colours (black, blue, green, cyan, red, magenta, yellow and white). The rows can independently select double height and/or double width and the start and stop positions of a background window of any colour. The signals sent to the TV are Red, Green, Blue, fast-blanking and half-tone. Separate horizontal and vertical synchronisation inputs are required.

The OSD architecture employed includes only a single line of display RAM. This makes the software more complicated but reduces the silicon area required to implement the OSD function. The software is required to update the display RAM on a regular basis. When operating in the 625-line PAL standard the updates must occur at 1.66 ms (26 lines) intervals in order to display adjacent lines. The OSD hardware can generate an interrupt when an update is required. There are 18 data registers (one for each character) and 6 control registers arranged as shown below. The table is for the T1, some of the control bits are different in the T4/7/8.

<table>
<thead>
<tr>
<th>$20-$31</th>
<th>OSD Data registers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$32 CAS</td>
<td>Read: status, Write: colours 1 &amp; 2 and outline enable.</td>
</tr>
<tr>
<td>$33 C34</td>
<td>Colours 3 &amp; 4.</td>
</tr>
<tr>
<td>$34 RAD</td>
<td>Row address, character size, int. enable, RGB invert.</td>
</tr>
<tr>
<td>$35 WCR</td>
<td>OSD &amp; PLL enable, Window enable and start column.</td>
</tr>
<tr>
<td>$36 CCP</td>
<td>Window colour and end column.</td>
</tr>
<tr>
<td>$37 HPD</td>
<td>Horizontal position, standard selection.</td>
</tr>
</tbody>
</table>
The OSD display is timed from an on-chip 14 MHz oscillator which is phase locked to the TV's line synchronisation pulses. The vertical synchronisation depends on the standard in use. Four standards are available (15.75 kHz/60 Hz, 31.5 kHz/120 Hz, 15.625 kHz/50 Hz and 31.25 kHz/100 Hz). The standard is selected by control bits in the T1/2 but is automatic in the T4 and the T7/8.

64 OSD characters are mask programmed along with the user ROM. The spacing and full size of the characters is the same as 8 x 13 (for 625-line standard). This allows continuous graphics. Half-dot interpolation hardware doubles the apparent resolution to produce smoother characters. A software selectable black outline (a half-dot wide) is also implemented in the hardware. Because the half-dot circuitry has to know the information for the next line of pixels, a 14th line is available in the character generator ROM to facilitate look ahead. The vertical height of a character is 26 lines (52 induding interlace) and the horizontal width is 2.27 μs (1/7 μs per half pixel).

SOFTWARE

There are several approaches to writing OSD software to operate with the single line architecture. The choice will affect the amount of ROM and RAM used. One principle is to have a separate interrupt routine for each type of row to be displayed. This method will use little RAM but will be inefficient in its use of ROM. The other approach is to write a single interrupt routine which transfers display information from a block of normal RAM to the display RAM as it is required for each new line. This method will be more ROM efficient but requires a RAM location for every display character. The amount of RAM used depends on the maximum amount of data which has to be displayed at any one time. The choice between these two methods will depend on the type of data to be displayed. The first method may be better if much of the displayed data is fixed. This could be, for example, a series of menus. The second method will however be more appropriate if the data is mostly variable. This will usually be the case in conventional TV applications.

This application note describes an implementation of the second of the above approaches. A block of RAM is used to contain a copy of all the data to be displayed. The size of this block can be changed to reflect the number of rows and the number of characters per row. The choice made in the example described here is 8 rows of 16 characters. This is slightly less than the maximum available and was chosen because the total number of characters (128) corresponds to the available page 1 RAM in the MC68HC05T1. The choice of 16 characters per row also slightly simplifies the software. The software allows any eight of the ten available rows to be used but only the first 16 of the 18 available characters. This choice does not prevent access to the right-hand-side of the screen as the display can be moved to the right under software control. The use of page 1 for the RAM does not incur any significant compromise in execution time. It also leaves free the page 0 RAM for the rest of the TV control software, which would be made less efficient if it had to use page 1 RAM, where direct addressing and bit manipulation instructions cannot be used. This choice slightly increases the ROM used by the OSD code, as 3-byte extended store instructions sometimes need to be used to write data to the RAM used for OSD characters.

The 1-byte indexed addressing mode can however be used in page 1. This addressing mode can access up to address $1FE and is made use of in the example software. For example the OSDCLR routine used to initialise RAM locations used for OSD employs a CLR DRAM-1,X instruction. DRAM is the start of page 1 RAM at $100 so DRAM-1 evaluates as $FF a 1-byte offset.

INTERRUPT ROUTINE

The OSD update interrupt routine (INLINE) shown in the program listing transfers data from page 1 RAM to display RAM each time an interrupt occurs. The first operation is to increment the pointer which selects the next row number. This pointer (OSDL) is subsequently used to transfer the appropriate data from page 1 RAM to the OSD RAM. So that any row number can be used the pointer selects the number from a table unique to each type of display. The appropriate table is determined by the value of LIND. The pointer is incremented until the corresponding row number is zero when the pointer is reset to zero. This allows any sub-set of up to 8 of the 10 available rows to be used. The next row number (ORed with the character size information contained in RAM) is written into the appropriate register ($34). The row number in this register is compared by the OSD hardware with the current position of the raster. When they match, an interrupt is generated and the next interrupt routine is performed. The other control registers are then updated from the page 0 RAM locations, which are used for this purpose.

To save RAM only three (RAD, CAS , & CCR) of the six control registers are loaded in this way. The pointer OSDL is multiplied by 3 using the table M3, as this is quicker than shifting and adding. In this example the other registers are loaded by the main program and therefore have fixed values for each display. The fixed registers are Colour 3/4 ($33), Window enable/start column ($35) and Horizontal position delay ($37). As this choice would not allow windows to be enabled on individual rows, window enable is controlled by the un-used bit (6) in the RAM byte used to update the Colour 1/2 register (CAS). This choice of fixed registers limits the flexibility of the display but clearly all registers can be updated on a line-by-line basis if more RAM is used. The limitations imposed by this choice are that colours 3 & 4, the window start column and the horizontal position apply to a whole display rather than to individual lines. In practice these constraints were not found to be significant restrictions for the displays required for TV use.
The interrupt routine then transfers the relevant OSD data from page 1 RAM into the OSD data registers. This is done using linear, repetitive code in order to minimise the time taken by the interrupt routine. The code uses 8 cycles (4 μs) for each byte transferred. Less ROM space would be utilised if a loop was employed but this would use 28 cycles per byte. The best choice depends on whether time or ROM use is more critical. The example code includes a cycle count to calculate the length of the interrupt routine. The time taken is 121 ± 4 μs. This includes the time taken by the interrupt itself. An alternative method of OSD data transfer (TOSD2) using a loop is included as comments in the listing. It would take an additional 165μs.

The last task performed by the interrupt routine is to control any character or window flashing. The software allows one or two characters (on a selected row) and one window (on the same or a different row) to be flashed at a rate determined by the MCU’s timer. This function could be performed outwith the interrupt routine in the main program and the time taken to perform it is not included in the figure given above.

**MAIN OSD PROGRAM**

The remainder of the OSD control program does not write directly to most of the display registers. It simply puts the required display and control information into the blocks of RAM allocated for this purpose, together with supplying the co-ordinates of any required flashing characters or windows. It must, however, write to the display control registers not updated by the interrupt routine; in this example these are $33$, $35$ and $37$. The program has 4 main parts. These are the idle, channel name table, program/channel number and analogue displays. The idle display applies when no transient display (e.g. program number and channel number or name) is on. The OSD idle condition is selectable between blank and a small program number at the bottom right hand corner of the screen.

The OSD example program (assembler listing included) is just part of the code required to control a TV set. This program was incorporated in HC05T1 software along with four other modules. These were the base module (idle loop, transient control, local keyboard, IR, IIC and reset), the tuning module (PLL, analogue and NVM control), the stereo module (stereoton and Nicam) and the Teletext module (FLOF level 1.5).

The microprocessor in a TV application will usually need to handle the reception of IR commands. Polled methods of IR reception are most effective if the time made unavailable to them by interrupts is minimised. It is for this reason that the illustrated OSD interrupt routine was written to execute as fast as possible. This is, however, not so much of a problem if the interrupt which processes this information is not serviced until several hundred microseconds later. The allowable size of this delay will of course depend on the IR protocol in use. The bi-phase protocol used with the example OSD software (transmitter chip: MC144105) has a minimum spacing of 1 ms between consecutive edges.

The next section describes the OSD features of this software. Some of the data used in the OSD is passed from other modules (particularly the tuning module). The same RAM allocation file was used in all modules so this part of the listing shows the locations used to pass data between them.

**OSD FEATURES PROVIDED IN THE EXAMPLE PROGRAM**

**Program change**

When keys 0-9, PC- or PC+ are pressed, the new program number appears (in cyan) at the bottom-right-hand corner of the screen in double height/double width characters and stays for 5 seconds after the last change. Above this display either the channel name (if one has been defined) or the channel number is shown (normal size). After 5 seconds this display times out and there is either no display or a permanent normal size program number display. This is selectable using the Teletext MIX key.

For program numbers of 10 and over, three keys are required. They are selected by first pressing “—”. Two flashing dashes are displayed, the first 0-9 key (only 0-4 valid) will be taken as the tens digit and the second as the units digit. If a new program number has not been selected within 30 seconds, the TV returns to the previous display (nothing or the old program number).

**Channel mode**

When the P/C key is pressed, the program number and channel name (or number) is displayed for 5 seconds as an indication of the current status. If it is pressed again during this period, the TV changes to channel mode. This will remain for 30 seconds after the last key-press. The display (in yellow) shows the program number as in program mode along with the channel number. The channel number flashes to show that it will be changed if a number or PC- or PC+ key is pressed. New channels can be selected. If the STORE key is pressed then the current channel is stored against the current program number. If no key is pressed for 30 seconds, the TV returns to program mode. If the channel has been changed but STORE not pressed then the TV will return back to the channel stored against the current program number.
Automatic search

When SEARCH is pressed the TV goes into the channel mode and the on screen display is as described above. The channel number is incremented at a rate of 2 per second until a signal is found. The search then stops. A press of STORE returns the TV to program mode, storing the new channel against the current program number.

Analogues

When any of the analogues are selected the appropriate logo is displayed along with a horizontal bar indicating the current value in the D/A convertor (full-scale 63). Display returns to default (nothing or program number) 5 seconds after the last change. If no analogue is selected the volume is shown (and adjusted) when the ANALOGUE +/- keys are used.

Channel name table

Up to 24 channels can have a 4 character name and standard bit associated with them. If the channel number and standard of one of these entries in the table correspond to those selected by the current program number then the name is displayed along with the program number when the program is selected or when P/C is pressed. Entry of names is done using the Teletext INDEX key. When it is pressed a table of six lines is displayed. Each line (identified by a "station" number in the leftmost column) contains a channel number, standard and the associated name. All of this data is user definable.

One character on the screen flashes to indicate the current position of the cursor. The character at the cursor position can be changed through 0-9, A-Z and space by pressing PC+ or PC- (0-9 for channel number digits and PAL/SECAM for standard). When a character (or the standard) is changed, its colour changes from yellow to red. The cursor is moved to the left and right by the Teletext RED and GREEN keys and up and down by the BLUE and YELLOW keys. The current line appears in a light blue (cyan) window as opposed to the dark blue window used for the other lines. The whole table scrolls when the cursor is required to go beyond the bottom (or top) of the current display.

To save a name the STORE key is pressed. This will save the name and standard on the current line against its channel number. This is indicated by the colour of any changed characters returning to yellow. Any changes which have been made to lines other than the one being stored are lost. Channel 00 cannot have a name. The procedure for removing a name from the table is to set the channel number to zero and then to save the line. Any name left on the line will not be used. The table display is exited by pressing the Teletext INDEX key. The function of each key is shown at the bottom of the display.
EXAMPLE PROGRAM

****************************
** TV/Teletext/OSD/Stereo program (MC68HC05T1/8). **
** On-Screen-Display module **
** This software was developed by Motorola Ltd. for demonstration purposes. **
** No liability can be accepted for its use in any specific application. **
** Original software copyright Motorola - all rights reserved. **
** P. Topping 25th May '90 **
****************************

IMPORT CHEX, CBEC, READ, WRITE, CDISP2
EXPORT NLINE, POSD, DRAM, ANOSD, PROSP, CHST
LIB RAMT1.505

SECTION.S RAM, COMM

********************************************************
** Teletext RAM allocation. **
********************************************************

SUB1 RMB 1 mode register
R1 RMB 1 page request address register
R2 RMB 1 page req. data reg. col. 0 : mg.
R3 RMB 1 " " " " 1 : pgt.
C1 RMB 1 " " " " 2 : pgu.
C2 RMB 1 " " " " 3 : ht.
C3 RMB 1 " " " " 4 : hu.
C4 RMB 1 " " " " 5 : mt.
C5 RMB 1 " " " " 6 : mu.

SUB2 RMB 1 display chapter register
R4 RMB 1 display control register (normal)
R5 RMB 1 display control register (news/sub)
R6 RMB 1 display mode register

SUB3 RMB 1 active chapter register
R7 RMB 1 active row register
R8 RMB 1 active column register
R9 RMB 1 active data register

PH RMB 1 2nd " "
PT RMB 1 3rd " "
P0 RMB 1 4th " "

LIFO RMB 9 LINKED PAGE No. LIFO BUFFER
PAGE RMB 7 PAGE No. INPUT BUFFER
PAG0 RMB 3 AC0 PAGE No.
PAG1 RMB 3 AC1 PAGE No.
PAG2 RMB 3 AC2 PAGE No.
PAG3 RMB 3 AC3 PAGE No.
PAGC RMB 3 CYAN PAGE No.
PAGI RMB 3 INDEX PAGE No.

PDF RMB 1 PAGE DIGIT POINTER
ACC RMB 4 DISP, RED, GREEN, YELLOW AC. CIR.

WACC RMB 1 WORKING ACC No.
ADD RMB 1 IIC ADDRESS
DPINT RMB 1 IIC DATA POINTER FOR WRITE

SUBAD RMB 1 IIC SUB-ADDRESS

I0BSUF RMB 4 IIC BUFFER, +2 & +3 RSVR FOR PLL

STAT2 RMB 1 0: ROW24 FETCH FLAG
  1: REMOTE REPEATING
  2: SEARCH/STANDBY IIC LOCK
  3: STANDBY STATUS
  4: UPDATE PENDING
  5: DIFFERENCE FOUND
  6: NO TELETEXT TRANSMISSION
  7: MIXED

LINKC RMB 1 LINK OPTIONS

STAT3 RMB 1 0: CYAN LINK ON
  1: YELLOW LINK ON
  2: GREEN LINK ON
  3: LINKS/ROW24 ON
General RAM allocation.

RAM allocation for Stereoton.

Poll timer
Tone (unadjusted for loudness)
Loudspeaker balance variable
Loudspeaker left volume
Loudspeaker right volume
Headphone volume left
Headphone volume right
Tone variable (Bass/Treble)
Current matrix
Present mode (mono/stereo/lan 1/11.12/12.11)
Workspace 1 (no interrupt usage)
Workspace 2 (for these please.)
Mono ident count
Stereo ident count
Dual lang ident count
Error ident count
Ident countdown
Total ident poll number
Temperature
<table>
<thead>
<tr>
<th>Address</th>
<th>RMB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000083</td>
<td>1</td>
<td>STAT5 RMB 1 0: LOUDNESS</td>
</tr>
<tr>
<td>0000086</td>
<td>1</td>
<td>CAS1 RMB 1 Row 1, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>0000087</td>
<td>1</td>
<td>RAD1 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>0000088</td>
<td>1</td>
<td>CCR1 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>0000089</td>
<td>1</td>
<td>CAS2 RMB 1 Row 2, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>000008a</td>
<td>1</td>
<td>RAD2 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>000008b</td>
<td>1</td>
<td>CCR2 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>000008c</td>
<td>1</td>
<td>CAS3 RMB 1 Row 3, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>000008d</td>
<td>1</td>
<td>RAD3 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>000008e</td>
<td>1</td>
<td>CCR3 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>000008f</td>
<td>1</td>
<td>CAS4 RMB 1 ROW 4, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>0000090</td>
<td>1</td>
<td>RAD4 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>0000091</td>
<td>1</td>
<td>CCR4 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>0000092</td>
<td>1</td>
<td>CAS5 RMB 1 Row 5, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>0000093</td>
<td>1</td>
<td>RAD5 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>0000094</td>
<td>1</td>
<td>CCR5 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>0000095</td>
<td>1</td>
<td>CAS6 RMB 1 ROW 6, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>0000096</td>
<td>1</td>
<td>RAD6 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>0000097</td>
<td>1</td>
<td>CCR6 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>0000098</td>
<td>1</td>
<td>CAS7 RMB 1 ROW 7, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>0000099</td>
<td>1</td>
<td>RAD7 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>000009a</td>
<td>1</td>
<td>CCR7 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>000009b</td>
<td>1</td>
<td>CAS8 RMB 1 ROW 8, colour 1/2 &amp; outline enable</td>
</tr>
<tr>
<td>000009c</td>
<td>1</td>
<td>RAD8 RMB 1 Row address &amp; character size</td>
</tr>
<tr>
<td>000009d</td>
<td></td>
<td>CCR8 RMB 1 Window colour &amp; end column</td>
</tr>
<tr>
<td>000009e</td>
<td>1</td>
<td>OSDL RMB 1 CURRENT OSD ROW POINTER</td>
</tr>
<tr>
<td>000009f</td>
<td>1</td>
<td>LIND RMB 1 ROW TABLE INDEX</td>
</tr>
<tr>
<td>00000a0</td>
<td>1</td>
<td>BROW RMB 1 CHARACTER FLASH ROW</td>
</tr>
<tr>
<td>00000a1</td>
<td>1</td>
<td>BCOL RMB 1 CHARACTER FLASH COLUMNS</td>
</tr>
<tr>
<td>00000a2</td>
<td>1</td>
<td>WROW RMB 1 WINDOW FLASH ROW</td>
</tr>
<tr>
<td>00000a3</td>
<td></td>
<td>*ROW1 RMB 1 FIRST ROW No. (NAME TABLE)</td>
</tr>
<tr>
<td>00000a4</td>
<td>1</td>
<td>ANAL RMB 1</td>
</tr>
<tr>
<td>00000a5</td>
<td>1</td>
<td>AANA RMB 1</td>
</tr>
<tr>
<td>00000a6</td>
<td>1</td>
<td>TEMP2 RMB 1</td>
</tr>
<tr>
<td>00000a9</td>
<td>22</td>
<td>STACK RMB 22 23 BYTES USED FOR STACK</td>
</tr>
<tr>
<td>00000bf</td>
<td></td>
<td>SP RMB 1 (1 INTERRUPT AND 9 NESTED SUBS)</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>KEI EQU $00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>KEYO EQU $00</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>KEIE EQU $00</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>SERO EQU $03</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>VOLU EQU $0A</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>CMET EQU $09</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>BRIL EQU $0C</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>SATU EQU $0D</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>L1 EQU $05</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>L2 EQU $06</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>WIDE EQU $04</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>PST EQU $05</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>VCR EQU $06</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>LOUD EQU $05</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>MUT EQU $03</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>STADD EQU $80</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>NORMVOL EQU $25</td>
</tr>
</tbody>
</table>

Key definitions:
- **KEYI** and **KEYO** are used for keyboard input and output.
- **KEIE** and **SERO** are used for keyboard enable and serial output.
- **VOLU**, **CMET**, **BRIL**, and **SATU** are used for volume control.
- **L1** and **L2** are used for language indicator bits.
- **WIDE** is used for wide matrix bit.
- **PST** is pseudo-stereo matrix bit.
- **VCR** is used for VCR active bit.
- **LOUD** is loudness effect active bit.
- **MUT** is mute indicator bit.
- **STADD** is stereo address (STC).
- **NORMVOL** is normal volume (mid balance).
Equate.

* PORTA EQU $00 Port A address
  PORTB EQU $01 Port B
  PORTC EQU $02 Port C
  PORTD EQU $03 Port D
  DDRA EQU $04 Port A data direction reg.
  DDRB EQU $05 Port B
  DDRC EQU $06 Port C
  DDRD EQU $07 Port D
  TCR EQU $12 Timer control register.
  TSR EQU $13 Timer status register.
  ICRH EQU $14 Input capture register, high.
  ICLR EQU $15 Input capture register, low.
  OCRL EQU $16 Output compare register, high.
  OCRH EQU $17 Output compare register, low.
  TDRH EQU $18 Timer data register, high.
  TDDR EQU $19 Timer data register, low.
  MISC EQU $1C Misc. register

OSD EQU $20 18 OSD data registers
CAS EQU $32 Color & status register
C34 EQU $33 Color 3/4 register
RAD EQU $34 Row address & character size
WCR EQU $35 Window/Column register
CGR EQU $36 Column/color register
HPD EQU $37 Horizontal position delay
MAD EQU $39 M-bus address register
MFD EQU $3A M-bus frequency divider
MCB EQU $3B M-bus control register
MSC EQU $3C M-bus status register
MCR EQU $3D M-bus data register
TR1 EQU $3E Test 1, OSD/Timer/PLM
TR2 EQU $3F Test 2, EPROM

SECTION .RAM2
DRAM RMB 128

SECTION .ROM2

OSD update routine - row number & data.

NLINE  LDA  OSDL  3  INC EMENT
INCA  STA  OSDL  3  6  LINE POINTER
STAG  STA  OSDL  4  10  27
ADD  LIND  3  13  30
TAX  2  15  32
LDA  LTAB0,X  5  20  37
BEQ  STAG  3  23  40  32 +/- 8
LDX  OSDL  3  LINE POINTER
LDA  M3,X  5  8  MULTIPLY BY 3
MBCA  OSL  4  CHARACTER SIZE INFO.
STA  RAD  4  16
MCR  OSL  4  20
MCR  CAS1,X  4  24
MCR  CAS  3  29
ROL  32
MCR  ROL  3  35
MCR  OF CASX
BCC  SKIPW  3  38
BEQ  7,WCR  5  43
LDA  CCR1,X  4  47
STA  CCR  4  51
LDA  M16,X  5  12
TOSD1 LDA <DRAM-16,X  4  USE IT TO ENABLE WINDOW
STA  OSD  4  8
LDA  <DRAM-15,X
STA  OSL
LDA  <DRAM-14,X
STA  OSD+2
LDA  <DRAM-13,X

Takes only 128 (8x16) Cycles.
83 0000039 b723 STA OSD+3
84 000003b >e6f4 LDA <DRAM-12,X
85 000003d b724 STA OSD+4
86 000003f >e65 STA <DRAM-11,X
87 0000041 b725 LDA OSD+5
88 0000043 >e66 STA OSD+6
89 0000045 b726 LDA <DRAM-10,X
90 0000047 >e67 STA OSD+7
91 0000049 b727 LDA <DRAM-9,X
92 000004b >e6f8 STA OSD+8
93 000004d b728 LDA <DRAM-8,X
94 000004f >e6f9 STA OSD+9
95 0000051 b729 LDA <DRAM-7,X
96 0000053 >e6fa STA OSD+10
97 0000055 b72a LDA <DRAM-6,X
98 0000057 >e6fb STA OSD+11
99 0000059 b72b LDA <DRAM-5,X
100 000005b >e6fc STA OSD+12
101 000005d b72c LDA <DRAM-4,X
102 000005f >e6fd STA OSD+13
103 0000061 b72d LDA <DRAM-3,X
104 0000063 >e6fe STA OSD+14
105 0000065 b72e LDA <DRAM-2,X
106 0000067 >e6ff STA OSD+15
107 0000069 b72f LDA <DRAM-1,X
140 00000aa 2142 WITH INT 242 (121 +/- 4 US)

******************************************************
* Character and window flash. *
******************************************************

*TOSD2 STX TFP1 4 ALTERNATIVE OSD DATA
* LDX #16 2 6 TRANSFER USING A LOOP.
* STX TFP2 4 10 THIS HAS THE ADVANTAGE
* OSDK LDX TFP1 3 OF USING 44 BYTES LESS
* LDA <DRAM-1,X 4 7 ROW BUT IT USES TWO MORE
* DEC TFP1 5 12 TEMPORARY RAM LOCATIONS
* LDX TFP2 3 15 AND TAKES 330 CYCLES
* STA OSD-1,X 5 20 (165us) LONGER.
* DEC TFP2 5 25
* BNE OSDK 3 28 28x16+10=458+128+330

109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126 000006b >09001f
127 000006d >b600
128 0000070 2719
129 0000072 6b34
130 0000074 a0f
131 0000076 >b100
132 0000078 2611
133 000007a >b600
134 000007c >a0f
135 000007e 97
136 000007f ad1b
137 0000081 >be00
138 0000083 54
139 0000084 54
140 0000085 54
141 0000086 54
142 0000087 2702
143 0000089 ad11
144 000008b 200e
145 000008d >b600
146 000008f 270a
147 0000091 b634
148 0000093 a0f
149 0000095 >b100
150 0000097 2602
151 0000099 1f35
152 000009b 80
153
154 000009c ec620
155 000009e a43f
156 00000a0 2609
157 00000a2 ec62 STA OSD,X
158 00000a4 a4c0 AND #53F
159 00000a6 ab0e BNE NTSP
160 00000a8 8720 STA OSD,X
161 00000aa 81 RTS
162 00000ab 6220 NTSP CLR OSD,X
163 00000ad 81

81
**OSD idle condition.**

---

**OSDEF**  
BRSET 3, STAT5, DOFF  
BSET 3, STAT5  
BRA OSDLLE

**DOFF**  
BCLR 3, STAT5

**OSDLE**  
SEI  
BCLR 4, STAT5  
NOT ANALOGS

LDX #29  
CLEAR PAGE 0

**DOOP**  
CLR CAS1-1, X  
OSD CONTROL

DECK  
BNE DOOP

**JSR**  
CDISP2

**CLR** $30

**BRSET** 3, STAT5, SKFDEF

**LDA**  
DRAM+12  
PROGRAM NUMBER

**LDA**  
DRAM+14

**STA** $31

**SKPDEF**  
CLR  
LDA #127

**BSR** OSDLLE

**LDA** #80010000  
HORIZONTAL POSITION: ZERO

**STA** HPD

**LDA** #10100011  
COLOR 1, 0 = RED, Cyan, EDGE ON

**BCLR** 2, STAT4, PMD  
PROGRAM MODE?

**LDA** #10100110  
NO, COLOR 0 = YELLOW

**PMD**  
STA CAS1

**LDA** #80010000  
SINGLE WIDTH/HEIGHT

**STA** RAD1

**LDA** #90C  
DEFAULT TO VOLUME

**STA** ANAL

**LDA** #AVOL

**STA** ANAP

**CLI**

**RTS**

**OSDCLR**  
IMCA  
STA W1

**DCLR**  
IMCA  
STA W1

**CLR** DRAM-1, X

**CPX** W1

**BNE** DCLR

**RTS**
**Program/Channel/Name main loop.**

```
*...
*...
*...

BNLP  LDA  W2          STATION No.
     STX  W3
     JSR  CBCD
     STA  W1
     AND  $0F
     ADD  $10
     LDX  W3
     STA  DRAM+1,X
     LDA  W1
     LSR
     LSR
     BNE  NOTZIR

NOTZIR  LDA  $F0
     ADD  $10
     STA  DRAM,X
     LDA  W2
     ADD  $DF
     STA  SUBADR
     LDA  $40
     STA  ADDR
     JSR  READ
     LDA  IOBUF+1
     STalion No.
     AND  $7F
     STA  W1
     AND  $0F
     STA  ADDR
     JSR  GNAME2

STATION   STATION   CHANNEL
     STATION   No.

ADD  $10

LEADING ZERO BLANK

LSR

BSET  STAT5

BRSET  7,IOBUF+1,PALS

BCLR  5,STAT5

PALS  JSR  CHGST

BSET  5,STAT5

BRESET  7,IOBUF+1,PALS

BCLR  5,STAT5

JMP  BNLFP

NOJMP  CLRX

MUL  LDA  MTAB,X
     CMP  $5C
     BEQ  ANFIN
     STA  DRAM+112,X
     INCX
     BPL  NTL
     BRA  NTL

ANFIN  JSR  WIND

SEC30  BSET  4,STAT

BLS  $30

STA  TMR

RTS

84
```
Look for channel name..
Cursor control (left & right).

CUTTAB FCB 3,4,12,13,14,15

CLFT BSR FCUR
CPX $5
BLO NRAP1
LDX $FF

NRAPl INCX
NEWC LDA CUTTAB,X
SEC32 JMP SEC30

CRGT BSR FCUR
TSTX
BNE NRAP2

NRAPl INCX
LCUR TAB,X
CMF W1
BNE CRNF

INCX

LOX $FF
INCX

LCUR TAB,X
CMF W1
BNE CRNF

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

INCX

IN
553
554
555
556
557

558

559 0000032c >b00
560 0000032e >b00
561 00000330 5a
562 00000331 5a
563 00000332 5a
564 00000333 >d0000
565 00000336 >d0006
566 00000339 a3f
567 0000033b a130
568 0000033d 2702
569 0000033f >d100
570 00000341 >d100
571 00000343 >d0000
572
573 00000346 a630
574 00000348 >d70006
575 0000034b a621
576 0000034d >d70007
577 00000350 a62c
578 00000352 >d70008
579 00000355 a600
580 00000357 >d70009
581 0000035d 03021c
582 00000360 >a0019
583 00000363 a933
584 00000365 >d70006
585 00000368 a625
586 0000036a >d70007
587 0000036d a623
588 0000036f >d70008
589 00000372 a621
590 00000374 >d70009
591 00000377 a62a
592 00000379 >d7000a
593 0000037c >d00012
594 00000380 >b00
595 00000382 >b700
596 00000384 >d60006
597 00000387 a04b
598 00000389 >d70006
599 0000038c 5c
600 0000038d >b300
601 0000038f 26f3
602 00000391 81
603
604
605
606
607
608
609
610
611
612
613
614 00000392 a4d0
615 00000394 4c
616 00000395 a63f
617
618 00000397 a119
619 00000399 2208
620 0000039b a110
621 0000039d 2410
622 0000039f a610
623 000003a1 200c
624 000003a3 a121
625 000003a5 2202
626 000003a7 a621
627 000003aa a13a
628 000003ab 2302
629 000003ad a600
630
631 000003af aa40
632 000003b1 >d70000
633 000003b4 >c0000
634
635 000003b7 a11b
636 000003b9 4a
637 000003ba a63f
638

******************************************************************************
*                                                                         *
*                          Standard change.                                  *
*                                                                         *
******************************************************************************

CHST     BCLR  5,STAT5  DEFAULT TO SECAM
LDX      BROW
DECX     DECY
LDX      M16,X
LDA      DRAH+6,X
AND      #38F
CMP      $30
BEQ      SIER
BSET     5,STAT5
          NO, MAKE IT PAL
            STANDARD CHANGED
JSR      SEC30

CHGST    LDA      $30
          STA      DRAH+6,X
          LDA      $23
          STA      DRAH+9,X
          STA      DRAH+10,X
          STA      DRAH+7,X
          LDA      $21
          STA      DRAH+9,X
          LDA      $2D
          STA      DRAH+10,X
          LDA      $63
          STA      DRAH+6,X
          LDA      $25
          STA      DRAH+7,X
          LDA      $23
          STA      DRAH+8,X
          LDA      $21
          STA      DRAH+9,X
          LDA      $2D
          STA      DRAH+10,X
          LDA      $30
          STA      DRAH+6,X
          ADC      $5
          STA      COUNT
          XLP      LDA      DRAH+6,X
          ADD      $40
          STA      DRAH+6,X
          INX      INX
          CFX      COUNT
          BNE      XLP
RTS

NSTCH    RTS
          **********************
          *                    *
          * Character change.  *
          *                    *
          **********************

PLUS     BSR     GETIT
         INCA
         AND      $3F
         CMP      $919
         BHI      MT9
         LTE9    CMP      $10
         BHS      NLTO
         LDA      $10
         BRA      NLTO
         MT9     CMP      $21
         BHI      MTA
         LDA      $21
         BRA      MTA
         MTA     CMP      $3A
         BLS      NLTO
         SPACE    LDA      $500
         SPACE
         NLTO    ORA      $40
         STA      DRAH,X
         JMP      SEC30
         MINUS    BSR     GETIT
         DECA
         AND      $3F

87
* ****************************************************** *
  * Name store (continued).                           *
  * ****************************************************** *

00000433 >be00  LDX  W3
00000435 >d60003  LDA  DRAM+3,X
00000438  48  LSLA
00000439  48  LSLA
0000043a  48  LSLA
0000043c >b700  STA  W1
0000043e >d60004  LDA  DRAM+4,X
00000441  a40f  AND  #$0F
00000443 >bb00  ADD  W1
00000445 >cd0000  JSR  CHEX
00000448 >b700  STA  IOBUF
0000044a >be00  LDX  W3
0000044c >d60006  LDA  DRAM+6,X
0000044e  a43f  AND  #$3F
00000451  a133  CMP  #$33
00000453  2702  BEQ  STSEC
00000455 >1e00  BSET  7,IOBUF
00000457 >b600  STSEC  LDA  COUNT
00000459 >bb00  ADD  BROW
0000045b  abdc  ADD  #$DC
0000045d >b700  STA  SUBADR
0000045f  a602  LDA  #$2
00000461 >b700  STA  W1
00000463 >b700  STA  W2-
00000465 >ae00  LDX  #$SUBADR
00000467 >cd0000  JSR  WRITE
0000046a >cd0000  JMP  FINBN
0000046d >cd0000  **
00000470  0a00  LTAB0  FCB  10,0  IDLE DISPLAY
00000472  090800  LTAB1  FCB  9,8,0  PR/CH DISPLAY
00000475  07080a00  LTAB2  FCB  7,8,10,0  ANALOGUE DISPLAY
00000479  0203040506070809  LTAB3  FCB  2,3,4,5,6,7,8,9,0  PR/CH/STD/NAME TABLE
00000482  1020304050607080  M16  FCB  $10,$20,$30,$40,$50,$60,$70,$80  MULT x 16
0000048a  000306090c0f1215  M3  FCB  0,3,6,9,12,15,18,21  MULT x 3

* ****************************************************** *
  * OSD line number tables.                          *
  * ****************************************************** *

89
Bottom corner Program/Channel no. display.
820
821
822
823
824
825
826 000004f4 0e121113
827
828
829 00000500 63b321f4f6efecf5
830
831 00000508 >b700
832 0000050a >b00041
833 0000050d >1800
834 0000050f 5f
835 00000510 a67f
836 00000512 >cd00
837 00000515 a6e6
838 00000517 >6ff
839 00000519 3f30
840 0000051a 26fb
841
842 0000051c a60a
843 0000051e b733
844 00000520 a6e6
845 00000522 b735
846 00000524 a622
847 00000526 b737
848 00000528 3f30
849 0000052a 3f31
850 0000052c a6e6
851 0000052e >b700
852 00000530 >b700
853 00000532 a6a6
854 00000534 >b700
855 00000536 a610
856 00000538 >b700
857 0000053a >b700
858 0000053c >b700
859 0000053e a6e3
860 00000540 >b700
861 00000542 >b700
862 00000544 a611
863 00000546 >b700
864 00000548 >3f00
865 0000054a a605
866 0000054c >b700
867
868
869
870
871
872
873
874
875 0000055e >b000
876 00000550 >600000
877 00000553 >700000
878 00000556 5c
879 00000557 >600000
880 0000055a >700001
881 0000055d 5c
882 0000055e >600000
883 00000561 >700010
884 00000564 5c
885 00000565 >600000
886 00000568 >700011
887

******************************************************

* Bottom row analogue bar. *

******************************************************

CHAR FCB $0E,$12,$11,$13 BARGRAPH CHARACTERS
ANCH FCB $63,$6F,$6E,$74,$A2,$82,$9A,$9C ANALOG
ANCH FCB $63,$B3,$21,$F4,$F6,$F7,$EC,$F5 LOGOS

****************************************************************

000004f8 636f6e74a2b2a9ac ANCH FCB $63,$6F,$6E,$74,$A2,$B2,$A9,$AC

****************************************************************

00000500 63b321f4f6efecf5 BARGRAPH CHARACTERS

****************************************************************

00000508 >b700

****************************************************************

0000050a >b00041

****************************************************************

0000050d >1800

****************************************************************

0000050f 5f

****************************************************************

00000510 a67f

****************************************************************

00000512 >cd00

****************************************************************

00000515 a6e6

****************************************************************

00000517 >6ff

****************************************************************

00000519 3f30

****************************************************************

0000051a 26fb

****************************************************************

0000051c a60a

****************************************************************

0000051e b733

****************************************************************

00000520 a6e6

****************************************************************

00000522 b735

****************************************************************

00000524 a622

****************************************************************

00000526 b737

****************************************************************

00000528 3f30

****************************************************************

0000052a 3f31

****************************************************************

0000052c a6e6

****************************************************************

0000052e >b700

****************************************************************

00000530 >b700

****************************************************************

00000532 a6a6

****************************************************************

00000534 >b700

****************************************************************

00000536 a610

****************************************************************

00000538 >b700

****************************************************************

0000053a >b700

****************************************************************

0000053c >b700

****************************************************************

0000053e a6e3

****************************************************************

00000540 >b700

****************************************************************

00000542 >b700

****************************************************************

00000544 a611

****************************************************************

00000546 >b700

****************************************************************

00000548 >3f00

****************************************************************

0000054a a605

****************************************************************

0000054c >b700

****************************************************************

0000055e >b000

****************************************************************

00000550 >600000

****************************************************************

00000553 >700000

****************************************************************

00000556 5c

****************************************************************

00000557 >600000

****************************************************************

0000055a >700001

****************************************************************

0000055d 5c

****************************************************************

0000055e >600000

****************************************************************

00000561 >700010

****************************************************************

00000564 5c

****************************************************************

00000565 >600000

****************************************************************

00000568 >700011

****************************************************************

****************************************************************

****************************************************************

LOGO LDX ANAL
LDX ANCH,X
ST A DRAM
INCX
LDX ANCH,X
ST A DRAM+1
INCX
LDX ANCH,X
ST A DRAM+16
INCX
LDX ANCH,X
ST A DRAM+17

****************************************************************

91
888
889
890
891
892
893
894 00000056b >b600
895 00000056d >3f00
896 00000056f 44
897 000000570 >3900
898 000000572 44
899 000000573 >3900
900 000000575 >b700
901 000000577 ae10
902 000000579 5a
903 00000057a >b300
904 00000057c 270a
905 00000057e 2204
906 000000580 a614
907 000000582 200d
908 000000584 a60e
909 000000586 2099
910 000000588 >bf00
911 00000058a >be00
912 00000058c >d60000
913 00000058f >be00
914 000000591 >d70020
915 000000594 5d
916 000000595 26e2
917 000000597 >cc0000
918 000000597 >cc0000
919 000000597 >cc0000
920 000000597 >cc0000

******************************************************
* Analogue bar. *
* ******************************************************

LDA  W3
CLR  W2
LSRA W2
ROL  W2
STA  W3
LDX  $16
LRA  DECK
CPX  W3
BEQ  STAR
BHI  DOT
LDA  $914 4
BRA  SKST
DOT  LDA  $10E
BRA  SKST
STAR STX  W1
LDO  W2
LDA  CHAR,X
LDA  $914 , 1, 2 OR 3
BDX  W1
SKST STA  DRAH+32,X
TSTX
BNE  LRA
JMP  SEC5
END
Serial bootstrap for the RAM and EEPROM1 of the MC68HC05B6

By Jeff Wright, Motorola Ltd., East Kilbride

INTRODUCTION

The MC68HC05B6 has 256 bytes of on-chip EEPROM, called EEPROM1, which can be used to store variable data in a non-volatile manner. In many applications this EEPROM1 will be used to hold a look-up table or system set up variables. In these cases it is usually a requirement that the EEPROM1 be initialised during the manufacture of the application. In addition, loading small programs into RAM and executing them is an easy way of trying out new software routines. This application note describes one method for serially loading (bootstrapping) the EEPROM1 via a program executing in the RAM of the MC68HC05B6.

BUILT IN BOOTSTRAP

The MC68HC05B6 has a built in RAM serial bootstrap program contained in the mask ROM of the device that uses the SCI. It would therefore seem a simple task to load programs into RAM; however, as ROM space on the device is obviously critical, a very simple protocol has been implemented. This means that the bootloader on the 'B6' does not accept S-records which are the normal output from an assembler; instead, the protocol expects pure binary data preceded by a count byte that holds the size of the program to be downloaded. No address information is contained in the download; instead, the bootloader always starts the program load at address $90$ in RAM. The first byte (the count byte) is stored here and then as the subsequent bytes are received via the SCI they are stored at incrementing RAM locations and the count byte is decremented for each byte received. When the count byte reaches zero the bootstrap program jumps to address $91$ and starts to execute the program that has just been loaded. No built in bootstrap routine is provided for the EEPROM1 array.

These restrictions present two problems:

i) How to convert assembler output to the format accepted by the 68HC05B6 RAM bootstrap routine?

ii) How to bootstrap the EEPROM1 of the 68HC05B6?

This application note provides a solution for each of these problems.
1) CONVERTING S-RECORDS FOR RAM BOOTSTRAP

To use the built-in RAM bootstrap program on the MC68HC05B6, the device must be configured as shown in Figure 1. If these conditions are met when the reset pin is released, then the serial bootstrap program described above will start to execute, and a program can be downloaded via a 9600 baud RS-232 source. Personal computers usually have one or more RS-232 ports referred to as COM ports. To overcome the format difference between S-records and that accepted by the bootloader, a conversion program is required. There is also an additional problem when using a PC—when a file is copied to a COM port to transfer it, it is the ASCII characters that are transmitted, not the binary data. This means for example that if a file containing the typed data byte $A5 was copied via the COM port to the B6, the B6 would in fact receive two bytes: $41 and $35, which represent the ASCII characters A and 5 respectively.

This means that the conversion program has to strip out the S-record format and convert the resultant data to binary format for transfer to the HC05B6. It must also insert the count byte at the beginning of the output file.

The pascal program BINCONV performs these three tasks; a listing of the source code is given at the end of this application note. A flow diagram of BINCONV can be seen in Figure 2. The inclusion of the count byte has been left as an option to increase the flexibility of the program, but it could easily be standardised to include the count byte for the B6 RAM bootloader. When BINCONV is invoked it prompts for the name of the S-record input file and the name required for the binary output file. After this each S-record in the input file is read and converted to binary data and stored in a temporary file. As each S-record is read it is echoed to the screen; when they have all been processed a message prompts the user and asks if a count byte is required. When used with the 68HC05B6 RAM bootloader the answer will always be yes, in which case the count value is written to the output file before the rest of the data is copied from the temporary file to the output file. Finally the value of the count byte is displayed for user confirmation—remember that the count byte is equal to the number of bytes in the program being converted plus 1 for the count byte itself. The program will only accept standard S-record format and will trap and abort if any non-valid character or format is detected.

With the PC COM port set for 9600 baud and the 68HC05B6 configured as in Figure 1 the binary file can be transferred and executed as follows:

i) Release Reset on the HC05B6

ii) Enter the command “COPY XXXX.YYY COM1\B” on the PC.

The program then be transferred to the B6 and execution started automatically. Note that the \B option is used to denote a binary file transfer so that the copy procedure does not abort if it finds an end of file (EOF) character in the middle of the file.
Figure 1. RAM bootstrap schematic
2) BOOTSTRAPPING THE EEPROM

To bootstrap the EEPROM1 on the MC68HC05B6 in the absence of a built in loader program, use must be made of the RAM bootloader described above. The idea is that an EEPROM1 loader can be written to the users exact requirements then assembled and downloaded into the RAM of the HC05B6 where it will execute and in turn download data and program it into the EEPROM1.

The 6K EEPROM emulation part, the MC68HC805B6, does have a built in EEPROM bootloader in place of the RAM bootloader and there is an accompanying PC program available from Motorola called E2B6 that downloads S-records to the device for programming.

The following is an explanation of an example EEPROM1 bootstrap program for the B6 that has been written to be compatible with the 805B6 PC program E2B6 thus eliminating the need to develop another PC program.

A listing of this program (EE1BOOT) is given at the end of this application note. The MC68HC05B6 has 176 bytes of RAM that can be used for the EEPROM1 bootstrap program, so the protocol must be kept simple and the code written efficiently. The format of the E2B6 program is a transfer of 2 address bytes followed by the data byte that is to be programmed at that location. At the same time the B6 returns the data from the previously programmed location for verification by E2B6. The program EE1BOOT has 4 main sections: a main loop, an erase routine, a program routine and an SCI service routine. The core of both the erase and program subroutines is the extended addressing subroutine EXTSUB which is used to access the EEPROM1 array. This subroutine is built in RAM by the main loop as the address information for the next byte to be programmed is received from the SCI. E2B6 always sends a null character during initialisation which could throw the EE1BOOT program out of synchronisation, as it is already executing before E2B6 is invoked. For this reason EE1BOOT ignores the first character received and treats the second as the first address byte.

The EXTSUB routine is first called as an “LDA $aaaa” to retrieve the last byte programmed for verification. Then the address in the routine is modified as the next address to be programmed is received. When the data byte is received the opcode of EXTSUB is incremented so that it becomes “STA bbbb” before the erase and program routines are called. After programming the opcode is decremented back to LDA before the main loop is repeated.

Note that the EEPROM1 location is always erased before programming. The timer output compare function is used to provide a 10ms delay for erasing and programming and the programming step is skipped to save time if the data presented to that location is $FF. The sequence of events to bootstrap the EEPROM1 of the 68HC05B6 is therefore as follows:

1) Configure the 68HC05B6 as in Figure 1.
2) Assemble the program EE1BOOT and convert it to binary using BINCONV as described in section 1.
3) Set up PC COM port to 9600 baud then release Reset on the HC05B6.
4) Use the command “COPY EE1BOOT.BIN COM1/B” to download EE1BOOT into the RAM of the HC05B6. EE1BOOT will now start to execute.
5) Start the program E2B6 on the PC and follow the instructions to download the desired S-records to the EEPROM1 of the 68HC05B6.

Note:

i) Only the download procedure of E2B6 will work in conjunction with EE1BOOT.

ii) Once the EEPROM1 security bit has been set, the RAM bootloader on the 68HC05B6 will no longer operate. This means that after the device has been reset it will be impossible to download any more data into the EEPROM1 until selfcheck has been executed – selfcheck performs an erase of the entire EEPROM1 array. This means that if the EEPROM1 is to be programmed in several steps, the one that will set the security bit should be done last.

FURTHER POSSIBILITIES

This application note has shown a method for initialising the EEPROM1 on the 68HC05B6 by using the RAM bootloader. It would of course be much simpler to incorporate a EEPROM1 bootloader in the ROM space of the user program, but often there is not enough space. If enough space is available (117 bytes), then EE1BOOT could be incorporated in the application software, thus saving steps 2, 3 & 4 in the procedure above.
Figure 2. Flow diagram of BINCONV
**EEBOOT - 68HC05B6 EEPROM Serial bootloader**

- This program is loaded into the RAM of the HC05B6 via the RAM bootloader. The program will then start to execute. The format has been selected to be the same as that on the 80586 so that the program 8286 can be used to program the EEPROM.

- Note: 8286 sends a null character during initialization so this program ignores the first character received on the SCI.

Jeff Wright Last Updated 10/5/90

---

**I/O and INTERNAL registers definition**

- I/O registers

```
0025 0000 PORTA EQU $00  port A.
0026 0001 PORTB EQU $01  port B.
0027 0002 PCRTC EQU $02  port C.
0028 0003 PORTD EQU $03  port D.
0029 0004 DDRA EQU $04  port A DDR.
0030 0005 DDRA EQU $05  port B DDR.
0031 0006 DDRC EQU $06  port C DDR.
0032
0033 0007 EECOUNT EQU $07
0034 0002 EIERA EQU 2
0035 0001 EILAT EQU 1
0036 0000 EIPCM EQU 0
0037
0038 000d BAUD EQU $0D
0039 000e SCCR1 EQU $0E
0040 0004 M8IT EQU 4
0041 000f SCCR2 EQU $0F
0042 0010 SCER EQU $10
0043 0005 RDRF EQU 5
0044 0011 SCDAT EQU $11
0045
0046  * TIMER registers
0047
0048 0012 TCR EQU $12  Timer control register.
0049 0005 TOIE EQU 5  Timer overflow interrupt enable.
0050 0006 OCIE EQU 6  Timer output compares interrupt enable.
0051 0007 ICIE EQU 7  Timer input captures interrupt enable.
```
0053 0013  TSR  EQU $13  Timer status register.
0054 0003  OCF2  EQU 3  Timer output compare 2 flag.
0055 0004  ICF2  EQU 4  Timer input capture 2 flag.
0056 0005  TOF  EQU 5  Timer overflow flag.
0057 0006  OCF1  EQU 6  Timer output compare 1 flag.
0058 0007  ICF1  EQU 7  Timer input capture 1 flag.
0059 0008  
0060 0016  TOC1HI  EQU $16  Timer output compare register 1 (16-bit).
0061 0017  TOC1LO  EQU $17  Timer output compare register 1 (16-bit).
0062 0018  TIMHI  EQU $18  Timer free running counter (16-bit).
0063 0019  TIMLO  EQU $19  Timer free running counter (16-bit).
0064 001a  
0065 001b  ***** MISC DEFINITIONS -----  
0066 001c  
0067 00c6  LDAEXT  EQU SC6  OP-Code for LDA extended.
0068 0014  MS10  EQU $14  10mS delay constant.
0069 0015  
0070 0016  
0071 0017  
0072 0018  
0073 0019  
0074 001a  START OF CODE      
0075 001b  
0076 001c  
0077 001d  
0078 0051  ORG $51  
0079 0052  
0080 0051  a6 00  RESET  LDA  #$00  %11 Ports inputs.
0081 0053  b7 04  STA  DDRA  
0082 0055  b7 05  STA  DDRB  
0083 0057  b7 06  STA  DDCR  
0084 0058  
0085 0059  19 0e  SCIINT  BCLR MBIT,SCCR1  Initialise SCI - 8 data bits.
0086 005b  a6 c0  LDA  #$2C  
0087 005c  b7 0d  STA  BAUD  9600 baud at 4MHz.
0088 005f  a6 c0  LDA  #$9C  Enable transmit and receive.
0089 0061  b7 0f  STA  SCCR2  
0090 0063  b7 10  STA  SCSR  Clear pending flags.
0091 0065  a6 c6  LDA  #$LDAEXT  Init extended addressing subroutine to LDA.
0092 0067  c7 00 8f  STA  OPCDE  
0093 006a  ad 1d  BSR  SCREAD  Wait here and ignore 1st char (E2B6 init).
0094 006c  ad 21  LOOP  BSR  EXTSUB  Load Acc with data from last programmed addr
0095 006e  b7 11  STA  SCDAT  Send it back for host to verify.
0096 0070  ad 17  BSR  SCREAD  Get high address
0097 0072  c7 00 90  STA  ADDHI  - and store it.
0098 0075  ad 12  BSR  SCREAD  Get low address
0100 0077  c7 00 91  STA  ADDLO  - and store it.
0101 007a  ad 0d  BSR  SCREAD  Get the data to be programmed
0102 007c  c7 00 93  STA  DATA  Store it temporarily.
0103 007f  3c 8f  INC  OPCDE  Change the ext addr subroutine to STA aaaa.
0104 0081  ad 11  BSR  ERASEE  Erase the selected address for 10mS.
0105 0083  ad 27  BSR  PROCEE  Now prog the data for 10mS.
0106 0085  3a 8f  DEC  OPCDE  Restore ext addr subroutine to LDA aaaa.
0107 0087  20 e3  BRA  LOOP  
0108 0051
SUBROUTINE TO SERVICE SCI

LDA SCDAT
RTS

EXTENDED ADDRESSING SUBROUTINE TO ACCESS FULL MEMORY MAP

DATA FCB 0 Reserved Byte for data during erasing.

ERASING SUBROUTINE

STA TIMLO Set timer for 10ms count
STA TSR - using output compare 1 function.

PROGRAMMING SUBROUTINE

Skip programming if data = $FF

DEL SET EIPOM,EECONT

SET-Up timer for 10ms count

- using output compare 1 function.

Wait here for end of erase time

- erase finished.

ADRLO FCB 0

BNL BSET E1LAT, EECONT

Set up timer for a 10ms count

- using output compare 1 function.

Wait here for programming to finish.
program BINCONV; { Program to convert Motorola S-record files to binary format. Optional inclusion of a count byte for HC05B6 RAM bootloader etc}
{ Programmer - Jeff Wright, MCU applications
Motorola
East Kilbride}
{ Last Updated 10/5/90}

{************************************************************************}

var
SrecFile : text;
BinFile : file;
Tempf : file;
srec : string[100];
Transfer : array[1..20000] of char;
numread, numwritten : word;
answer : char;
fnamei : string[15];
fnameo : string[15];
bytout : char;
countbyt : integer;
datcnt : integer;
datval : integer;
point : integer;
cnt1 : integer;
cnt2 : integer;
quit : boolean;
Count : boolean;

Procedure Calc_hex(chr1, chr2 : integer);
{Combines 2 characters into a single byte value i.e A5->165, error signaled if non hex character detected}

Begin
Case chr1 of
48..57 : chr1 := chr1 - 48;
65..70 : chr1 := chr1 - 55; { Is this a valid hex character?}
else
   begin
      writeln ('invalid data - conversion aborted');
      quit := true
   end
end;
Case chr2 of
48..57 : chr2 := chr2 - 48;
65..70 : chr2 := chr2 - 55;
else
   begin
      writeln ('invalid data - conversion aborted');
      quit := true
   end

101
datval := chr1*16 + chr2;  \{Convert to single byte\}
end;

Procedure Binwrite(length, dpoint : integer):
\{Converts an S-record line to hex and stores it in a temporary file\}
begin
  length := length-3; \{Allow for address and checksum bytes\}
  countbyt := countbyt+length; \{Update running byte total\}
  length := length*2; \{Twice as many characters as bytes\}
  while length > 0 do
  begin
    cntl := Ord(srec[dpoint]); \{Get the next two characters\}
    cnt2 := Ord(srec[dpoint+1]);
    dpoint := dpoint+2; \{Update pointer and length\}
    length := length-2;
    Calc_hex(cntl, cnt2); \{Convert two characters into single byte\}
    bytout := Chr(datval); \{- now convert that single byte into a \}
    blockwrite (tempf, bytout, l) \{character and save it in temporary file\}
  end
end;

begin
  writeln ('S-record to Binary conversion utility');
  writeln;
  write('Input S-record file name? -> '); readln(fnamei);
  assign(SrecFile, fnamei);
  writeln;
  write('Binary output file name? -> '); readln(fnameo);
  assign(BinFile, fnameo);
  assign(tempf, 'temp.tmp');
  quit := false;
  countbyt := 1;
  Reset(SrecFile); \{open the two \}
  Rewrite(BinFile, 1); \{ -selected files\}
  Rewrite(tempf, 1); \{ + a temporary file\}
while not Eof(SrecFile) and not quit do
begin
readln(SrecFile, srec);  { read S-rec into char string srec}
writeln(srec);
If srec[1]='S' then
begin
CASE srec[2] of
  '1' :
  begin
    cntl := Ord(srec[3]);  {get the 2 record length}
cnt2 := Ord(srec[4]);  {characters}
    calc_hex(cnt1,cnt2);  {func to produce hex in
    datcnt from cnt1 & 2}
datcnt := datval;
    point := 9;  {point to first data character}
    binwrite(datcnt,point)  { convert the data in this s-rec
    line to binary and store in temp file}
  end;
  '0' : writeln ('Conversion started');
  '9' : writeln ('last S-record done');
else
  begin  {If not S0,S1orS9 record then abort}
    quit := true;
    writeln ('Non standard S-record detected - Conversion aborted')
  end
end
end else
begin  {If 1st char not an S then abort}
    quit := true;
    writeln ('Non standard S-record detected - Conversion aborted')
end
end;
If quit = false then
{If no errors then copy the temporary file to the output file and add in
a count byte if required}
begin
  Reset (tempf,1);
  writeln;
  write ('Do you want a count byte added to start of output file? -> ');
  readln (answer);
  If upcase(answer) = 'Y' then
  Begin
    writeln ('Total size including count byte = ',countbyt);
    bytout := chr(countbyt);
    blockwrite (binfile,bytout,1)
  end;
repeat
  blockread (tempf,transfer,sizeof(transfer),numread);
  blockwrite (binfile,transfer,numread,numwritten);
until (numread=0) or (numwritten <> numread)
end;
close(tempf);
erase(tempf);  {Finished with temporary file so erase it}
close(SrecFile);
close(BinFile)  {Close files before quiting
end.
INTRODUCTION

An increasing number of applications involving MC68HC05 MCUs require large amounts of critical data to be stored in EEPROM memory. This application note describes software routines, generated for the HC05, which allow stored data to be encoded so that single bit errors existing in retrieved data may be corrected and two bit errors detected. The routines use a simple Linear Block Code for the encoding of stored data.

SINGLE BIT ERROR CORRECTION

All methods of error detection/correction involve the use of extra check bits added to the data bits to produce some form of codeword. To allow the detection of a single bit error in a specific codeword it is necessary that each word differs from any other word by at least two digits. A one bit error will then produce an invalid word. The number of digits by which two words, of the same length, differ is defined as the Hamming Distance. For the correction of up to \( t \) errors a minimum Hamming Distance of \( 2t + 1 \) is required between each codeword. Single bit error correction and double bit error detection requires a minimum distance of 3. The problem is to decide what an original codeword was if an invalid codeword has been detected. One means of doing this is to use a Linear Block Code, as described below. Linear Block Codes for the correction of single bit errors are referred to as Hamming Codes. The following describes a systematic method for single bit error correction.

A codeword consists of \( k \) data digits to which are added \( r \) check digits to produce an \( n \) digit codeword \( (n = k + r) \). The \( r \) data digits are redundant, in that they carry no additional data, and the code efficiency is defined as \( k/n \). This is an indication of the amount of information transferred, relative to the total number of bits.

For a linear block code the general codeword can be written in the form:

\[ a_1 a_2 a_3 \ldots \ldots a_k c_1 c_2 \ldots \ldots c_r \]

where \( a_1 \) to \( a_k \) are the \( k \) data digits and \( c_1 \) to \( c_r \) are the \( r \) check digits.

The check digits are chosen to satisfy the \( r \) linear equations:

\[ 0 = h_1 a_1 \oplus h_1 a_2 \oplus \ldots \ldots \oplus h_1 a_k \oplus c_1 \]

\[ 0 = h_2 a_1 \oplus h_2 a_2 \oplus \ldots \ldots \oplus h_2 a_k \oplus c_2 \]

\[ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \]

\[ 0 = h_r a_1 \oplus h_r a_2 \oplus \ldots \ldots \oplus h_r a_k \oplus c_r \]

Each element in the above equations is either a one or a zero and all addition is modulo 2.

These equations can be more conveniently expressed in terms of the matrix equation:

\[ [H][T] = 0 \]
where [T] is an $n \times 1$ column vector representing the stored codeword:

$$[T] = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ . \\ . \\ a_k \\ c_1 \\ c_2 \\ . \\ . \\ c_t \end{bmatrix}$$

and $[H]$ is an $r \times n$ matrix, referred to as the parity check matrix.

$$[H] = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1k} & 0 & \cdots & 0 \\ h_{21} & h_{22} & \cdots & h_{2k} & 0 & \cdots & 0 \\ . \\ . \\ h_{r1} & h_{r2} & \cdots & h_{rk} & 0 & \cdots & 1 \end{bmatrix}$$

A second column vector $[R]$, with same dimensions as $[T]$, is used to represent the retrieved codeword. This may or may not be equal to the original stored codeword $[T]$, depending on whether or not an error exists. If $[H][R] = 0$, then $[R]$ is most likely to be the original stored codeword. If $[H][R]$ gives a non zero value then at least one error has occurred. If an $n \times 1$ error matrix $[E]$ is introduced, then the retrieved codeword $[R]$ can be written as:

$$[R] = [T] + [E]$$

If $[E]$ consists totally of zeros then no error has occurred. For any error that does occur in $[R]$, $[E]$ will contain a ‘1’ in the corresponding position. The problem is then to determine where in $[E]$ the non zero elements are, once the codeword $[R]$ has been retrieved. A matrix $[S]$, referred to as the syndrome, is defined such that:

$$[S] = [H][R]$$

This can be expanded to

$$[S] = [H][T] + [H][E]$$

giving

$$[S] = [H][E]$$

$[S]$ is an $r \times 1$ column matrix and can consists of any one of $2^r$ sequences. $[E]$ is an $n \times 1$ matrix and can consist of any one of $2^n$ sequences. As $n > r$ there is no unique solution to the above equation. However, in this case it is assumed that only one error has occurred and therefore $[E]$ contains only one non zero element. Multiplying $[E]$ by $[H]$ yields a syndrome which will be equal to one column within $[H]$. The position of this column will indicate where the non zero element exists in $[E]$ and hence the position of the single bit error in $[R]$. In the case of two or more non-zero elements in $[E]$ error correction is not possible.

**HAMMING BOUND AND CODE EFFICIENCY**

The Hamming Bound is defined as:

$$2^r \geq k + (r + 1)$$

where $k$ is the number of data bits and $r$ is the number of check bits.

This must be satisfied for single bit error correction. To allow double bit error detection a further check bit must be added. Table 1 shows the number of check bits required, along with the corresponding code rate, for single bit error correction and double bit error detection in different numbers of data bits.

It can be seen from the table that, in general, the greater the number of data bits the greater the code efficiency. However as the size of the codeword increases the calculations involved in detecting an error become increasingly more cumbersome. It can also be seen that for both 8 and 11 data bits, the number of check bits required is 5. By using 11 data bits and 5 check bits the Hamming bound can be satisfied exactly. There is no exact solution when 8 data bits are used. However, this is a more convenient data size for an 8-bit MCU and is therefore used in this application, despite the lower code efficiency.

<table>
<thead>
<tr>
<th>No. of Data Bits (k)</th>
<th>No. of Check Bits</th>
<th>Code Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>61.5%</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>68.7%</td>
</tr>
<tr>
<td>26</td>
<td>6</td>
<td>83.9%</td>
</tr>
</tbody>
</table>

**Table 1. Check Bit Requirements and Code Efficiency for Single Error Correction**
CODEWORD GENERATION AND STORAGE

For one byte of data, 4 check bits are required for single bit error correction. The parity check matrix will consist of 12 columns of 4 bits and can be simply generated by taking the binary values $1$ to $C$ (represented as binary column vectors) to generate 12 columns as shown. The check bits, c1 to c4, are assigned to the columns containing a single non zero entry and the data bits , b7 to b0, are assigned to the remaining columns. The order of assignment is completely arbitrary.

\[
\begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

The following equations can then be derived from the parity check matrix and used to calculate c1 to c4.

\[
c_1 = b_7 \oplus b_6 \oplus b_4 \oplus b_3 \oplus b_1
\]

\[
c_2 = b_7 \oplus b_5 \oplus b_4 \oplus b_2 \oplus b_1
\]

\[
c_3 = b_6 \oplus b_5 \oplus b_4 \oplus b_0
\]

\[
c_4 = b_3 \oplus b_2 \oplus b_1 \oplus b_0
\]

At no time is the application software required to carry out any matrix multiplication. This is done implicitly by the use of the above equations. A fifth check digit, c5, is used to detect the occurrence of a double bit error and is a simple parity check (even parity) for the 12 bit codeword formed by concatenating b7–b0 and c1–c4.

Figure 1 shows the data organisation in memory. The data bytes (b0–b7) and the corresponding check bits (c1–c5) are stored separately in adjoining blocks of EEPROM. This allows executable code to be stored in the EEPROM and protected using error checking. The data EEPROM block is 256 bytes long. It is immediately followed by the check EEPROM block. The minimum size possible for the check EEPROM is 160 bytes (256 x 5 bits). In order that all check bits can be accommodated within this, software routines are required for the ‘packing’ and ‘unpacking’ of check bits.

DATA

<table>
<thead>
<tr>
<th>DATA</th>
<th>DATA EEPROM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>256 BYTES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATA + $100</th>
<th>CHECK EEPROM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>160 BYTES</td>
</tr>
</tbody>
</table>

Figure 1. Data Organisation in Memory

DATA RETRIEVAL AND CORRECTION

To allow a retrieved codeword to be checked it is necessary to generate the syndrome [S]. To do this the retrieved data byte is used to generate a new set of check bits, c1'–c4', using the same set of equations as above. The syndrome is then generated by exclusive ORing c1' to c4' with c1 to c4. An non zero result will indicate the presence of an error. The parity check c5' is calculated from the retrieved data and check bits and compared with c5. If they are the same, and the syndrome indicates the presence of an error, then it is assumed that a double error has occurred and can therefore not be corrected. If the error is correctable then the syndrome can be compared with values corresponding to the columns of [H] (in this case, a simple lookup table in ROM) to determine the error position.

SOFTWARE

An assembled listing of the software is included at the end of this application note.

The software has been written to run on the MC68HC05SC21 but can be easily modified to run on any HC05 MCU with EEPROM. It comprises 2 main routines. The first routine is CHECKPROG and this generates the codeword from the data and programs the data and the appropriate check bits into EEPROM.
The second routine, GETCHECK, retrieves the data and check bits from the EEPROM, calculates the syndrome and, if any error is detected, returns with the error position indicated in the accumulator. The detection of a double bit error by GETCHECK is indicated by the carry bit being set on return from the routine. The data and check EEPROM blocks can be placed anywhere within the device EEPROM memory, the start address of the data EEPROM being determined by an address held in RAM registers EPSTHI and EPSTLO.

Four further subroutines are called by the main routines. PAKCHK and UNPAKCHK are used for the packing and unpacking of the check bits in the check EEPROM block. CHECKBIT is used to calculate the check bits c1 to c4 and c1' to c4'. CALC5 calculates the parity checks c5 and c5'.

The total ROM requirement for the routines is 301 bytes with a further 56 bytes required for the EEPROM write/erase routines. Execution time for the routine GETCHECK is approximately 0.6 ms (with 2 MHz internal bus frequency). The execution time for CHECKPROG is dependent on the EEPROM programming time. The time required for the calculation and packing of the check bits amounts to approximately 0.6 ms.

REFERENCES

This software was developed by Motorola Ltd. for demonstration purposes only. Motorola does not assume liability arising out of the application or use of this software and does not guarantee its functionality.

Original software copyright Motorola - all rights reserved.

These routines use a modified (12,8) Hamming code to provide single bit error correction and double bit error detection for data stored in EEPROM. The data is segmented into blocks of 256 bytes. Each 256 block of 'data' EEPROM is immediately followed by 160 bytes of 'check' EEPROM which contains the parity check bits.

BYTE EQUATES

** USER EQUATES

** BIT EQUATES

PORT A SERIAL I/O PORT

SERIO EQU 0 Serial i/o port - port A bit 0

* PORT A SERIAL I/O PORT

MISC Register

ROMPG EQU 7

INTFF EQU 6

DCTST EQU 4

109
**Program Control Register**

WE EQU 7
VPON EQU 2
PGE EQU 1
PLE EQU 0

*************************************************************************
ORG ERRCOR
*************************************************************************

**CHKPROG** - This routine programs a byte of data held in acc. into a block of data EEPROM. Data EEPROM is 256 bytes long and starts from an address held in EPSTHI and EPSTLO. Location of data byte within data EEPROM is determined by X-reg value. Check bits C1 to C4 are calculated for the data byte, using a (12,8) block code, to allow the correction of a single bit error by the routine GETCHECK. A further simple parity check bit, C5, is generated to allow the detection of double bit errors. The check bits are programmed as a 5 bit block into the check EEPROM, which is 160 byte long and starts from location EPSTHI,EPSTLO + $100.

Enter with data to be programmed in acc., start add. of data EEPROM in EPSTHI and EPSTLO and index value for data address in X-reg.

Returns with X-reg. value saved.

<table>
<thead>
<tr>
<th><strong>CHKPROG</strong></th>
<th><strong>STA</strong></th>
<th><strong>DATA</strong></th>
<th>Store data byte</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STX</strong></td>
<td><strong>INDEX</strong></td>
<td><strong>SAVE data address index value</strong></td>
<td></td>
</tr>
<tr>
<td><strong>LDA</strong></td>
<td><strong>EPSTHI</strong></td>
<td><strong>Set up address offset for EPRWRT</strong></td>
<td></td>
</tr>
<tr>
<td><strong>STA</strong></td>
<td><strong>EPRADH</strong></td>
<td>- (EEPROM write routine)</td>
<td></td>
</tr>
<tr>
<td><strong>LDA</strong></td>
<td><strong>EPSTLO</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>STA</strong></td>
<td><strong>EPRADL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LDA</strong></td>
<td><strong>DATA</strong></td>
<td><strong>Restore data byte into acc.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>JSR</strong></td>
<td><strong>EPRWRT</strong></td>
<td><strong>Store data byte in EEPROM at address specified by EPSTHI,EPSTLO + x reg.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>JSR</strong></td>
<td><strong>CHECKBIT</strong></td>
<td><strong>Calculate check bits C1 to C4. Returns C1 to C4 in CHECK0 (bits 0 - 3)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>JSR</strong></td>
<td><strong>CALC5</strong></td>
<td><strong>Calculate C5 and return with C5 in CHECK0(4)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>JSR</strong></td>
<td><strong>PAKCHK</strong></td>
<td><strong>Calculate offset req'd to give byte location for C1 to C5 and store in X-reg. and rotate CHECK0 and CHECK1 so that C1 to C5 will be programmed into appropriate part of check EEPROM</strong></td>
<td></td>
</tr>
<tr>
<td><strong>INC</strong></td>
<td><strong>EPRADH</strong></td>
<td><strong>Increment start add. of data EEPROM by $100 to get start add. of check EEPROM</strong></td>
<td></td>
</tr>
<tr>
<td><strong>LDX</strong></td>
<td><strong>INDEX</strong></td>
<td><strong>Restore X-reg. value</strong></td>
<td></td>
</tr>
</tbody>
</table>
GETCHECK - Retrieves a data byte from location EPSTHI, EPSTLO + x along with corresponding check bits C1 to C5. The data, and C1 to C4, are used to calculate the SYNDROME value which is used to indicate the position of a single bit error. The SYNDROME value is generated by calculating new check bit values C1' to C4' from the retrieved data and adding these (modulo 2) to the original check bit values retrieved from the Check EEPROM. C5 and C5' are simple parity check bits used to indicate the occurrence of a 2 bit error.

Enter with start address of data block in EPSTHI and EPSTLO and index value for data byte in X-reg.

Returns with the uncorrected data byte in RAM location DATA. Error status is indicated by the following:

No errors - carry = 0, acc. = 0.

Single bit error in data byte - carry = 0, acc. has one bit set to indicate the position of the error in the data byte.

Single bit error in check bits - carry = 0, acc has upper nybble = $F and one bit set in lower nybble to indicate check bit error position. (b0 indicates error in C1, b3 indicates error in C4).

Double bit error - carry = 1.

X-reg. contents are saved.

*******************************************************************************
GETCHECK EQU *

STX INDEX Store data address offset

LDA EPSTHI Set up address offset for GETBYTE

STA EPRADH

LDA EPSTLO

STA EPRADL

JSR GETBYTE Get data byte from loc'n EPSTHI, EPSTLO + X

STA DATA Store retrieved data byte

JSR UNPAKCHK Set check bits and return with C1 to C5 in CHECK3 (0 to 4)

STA CHECK3

JSR CHECKBIT Calculate new check bits (C1' to C4') from retrieved data byte and return with them in CHECK0(0:3)

LDA CHECK0 Store C1' to C4'

STA CHECK4

LDA CHECK3

STA CHECK0

LDA CHECK1

STA CHECK2

LDA CHECK2

STA CHECK3

JSR CALC5 Calculate new parity check bit C5' from retrieved data and check bits (C1 to C4) and return with C1 to C4 in CHECK0(0 to 3) and C5' in CHECK0 (bit 4)

LDA CHECK3 Load C1 to C5

EOR CHECK4 Generate Syndrome

AND #$0F Mask out C5 from syndrome

BEQ NOERRIf syndrome = 0 then no error

STA SYNDROME Store syndrome
0204
0205
0206 1156 b6 94
0207 1159 a4 10
0208 115a 26 05
0209 115c 09 96 0e
0210 115f 20 03
0211
0212 1161 08 96 09
0213
0214
0215
0216
0217 1164 be 9b
0218 1166 d6 11 7b
0219 1169 98
0220
0221 116a be 93
0222 116c 81
0223
0224 116d 99
0225
0226 116e be 93
0227 1170 81
0228
0229
0230 1171
0231
0232 1171 a6 d6
0233 1173 b7 80
0234 1175 a6 81
0235 1177 b7 83
0236 1179 bd 80
0237 117b 81
0238
0239 117c f1
0240 117d f2
0241 117e 80
0242 117f f4
0243 1180 40
0244 1181 20
0245 1182 10
0246 1183 f8
0247 1184 08
0248 1185 04
0249 1186 02
0250 1187 01
0251
0252
0253
0254
0255
0256
0257
0258
0259
0260
0261
0262
0263
0264
0265
0266
0267 1188 be 93
0268 118a a6 05
0269 118c 42
0270
0271

LDA CHECK0  Load Cl to C4 and C5' into acc.
AND $10     Mask out Cl to C4 - leave C5'
BNE CORR1   Branch if C5' = 1
BCLR 4,CHECK3,DOUBLERR If C5 = C5' = 0 then 2 bit error exists
BRA FNDERR  If C5 = 1 and C5' then correctable
1 bit error exists'
CORR1 BXSET 4,CHECK3,DOUBLERR If C5 - C5' then 2 bit error exists'
If CS - C.S' - 0
then 2 bit error exists
If C5 - CS' - 0
then correctable
1 bit error exists'

FDERR LDX SYNDROME
Set bit in acc. to indicate pos. of error
Clear carry to indicate that double bit
error did not occur
LDX INDEX Restore X-reg. value
RTS

DOUBLERR SEC
Set carry to indicate double error
occurred
LDX INDEX Restore X-reg. value
RTS

GETBYTE EQU *
RAM Subroutine to load acc. with byte held
in EPRADH,EPRADL + x

BITPNT FCB $F1 Error in C1
FCB $F2 Error in C2
FCB $F0 Error in b7
FCB $F4 Error in C3
FCB $F0 Error in b6
FCB $F0 Error in b5
FCB $F0 Error in b4
FCB $F8 Error in C4
FCB $F0 Error in b3
FCB $F0 Error in b2
FCB $F0 Error in b1
FCB $F0 Error in b0

*******************************************************************************

UNPAKCHK - Calculates the address of the the 1st byte location
for the check data Cl to C5, for the data byte held in location
EPSTHI,EPSTLO + INDEX, and then fetches the two bytes containing Cl
to C5 and stores them in CHECK1 and CHECK0. These locations are then
rotated left until Cl is located in the lsb position of CHECK0.
Enter with start address of DATA EEPROM in EPSTHI and EPSTLO
and address index in RAM reg. INDEX.
Returns with Cl to C5 in CHECK0 (0 - 4). Acc. and X-reg. contents
not saved.
*******************************************************************************

UNPAKCHK LDX INDEX
LDA $905 Multiply DATA offset by 5 and store
result in acc. and x-reg.

************************************************************************************
CLR     REM         Initialise remainder RAM reg.

Divide result in X-reg. and acc. by 8 and store remainder in RAM byte REM

0277 118f 56  RORX
0278 1190 46  RORA

0283 1197 24 02  BCC    UPCHKAD2
0284 1199 12 98  BSET   1,REM

0285 119b 56  RORX
0286 119c 46  RORA

UPCHKAD2 RORX

UPCHKAD1 RORX

0289 11af 3d 98  TST     REM

0290 11a1 97  UPCHKAD3 TAX  Store index for checkbits in X-reg.

0291 11a2 3c 81  INC     EPRADH  Inc. GETBYTE address offset by $100

0292 11a4 cd 11 71  JSR    GETBYTE Get 1st byte of C1 to C5 string

0293 11a7 b7 94  STA    CHECK0

0294 11a9 5c  INX

0295 11aa cd 11 71  JSR    GETBYTE Get 2nd byte of C1 to C5 string

0296 11ad b7 95  STA    CHECK1

0297  
0298  

UPCHKAD4 CLC

0300 11b1 27 09  BEQ    UPNOREM

0301 11b3 98  

0302 11b4 36 95  ROR    CHECK1 - the no. of left shifts is equal to the value held in REM

0303  

0304  

0305 11b6 36 94  ROR    CHECK0 - msb of CHECK0 is shifted into lsb of CHECK1.

0306 11b8 3a 98  DEC    REM

0307 11ba 26 f7  BNE    UPCHKAD4

UPNOREM LDA    CHECK0

0309 11bc b6 94  AND    #$1F Mask out non valid check bits

0310 11be a4 1f  

0311 11c0 81  RTS

CHECKBIT - Calculates the checkbits C1 to C4

Enter with data in RAM reg. DATA

Returns with C1 to C4 in CHECK0 (0 - 3).

Acc., X-reg. contents not saved.

***************************************************************************************************************************

CLR     CHECK0     Clear CHECK0 reg.

LDA     DATA      Load data byte into acc.

AND    #$00001111  Load data byte

(c4 = b3 + b2 + b1 + b0 (mod 2)

(mask out other bits)

BSR    CALCHK  Calculate c4 (result returned in carry)

ROL    CHECK0  rotate c4 into CHECK0

LDA     DATA      Load data byte

AND    #$01110001 (mask out other bits)

BSR    CALCHK  Calculate c3 (result returned in carry)

ROL    CHECK0  rotate c3 into CHECK0

LDA     DATA

Load data byte

113
AND #10101010  c2 = b7 + b5 + b4 + b2 + b1 (mod 2)  
      (mask out other bits)
BSR CALCHK  Calculate c2 (result returned in carry)
ROL CHECK0  rotate c2 into CHECK0

LDA DATA  Load data byte
AND #1101110  cl = b7 + b6 + b4 + b3 + b1 (mod 2)  
        (mask out other bits)
BSR CALCHK  Calculate c1 (result returned in carry)
ROL CHECK0  rotate c1 into CHECK0

CALC5 - Calculates the parity checkbit C5

Enter with data in acc. and C1 to C4 in CHECK0 (0 to 3).
Returns with C5 in CHECK0 (bit 4).
Acc., X-reg. contents not saved.

CALC5  LDA DATA
BCLR 4,CHECK0  Clear C5
BSR CALCHK  Calc. C' = b7+b6+b5+b4+b3+b2+b1+b0 (mod 2)
BCC NOSETC5
BSET 4,CHECK0  Set C5 if C = 1

NOSETC5  LDA CHECK0
BSR CALCHK  Calc. c5 = C'+c1+c2+c3+c4
BCS SETC5
BCLR 4,CHECK0  Clear C5 if total no. of 1s in DATA and 
c1 to c4 is even ((DATA,c1 to c4) parity 
is even)

SETC5  BSET 4,CHECK0  Set C5 if (DATA,c1 to c4) parity is odd.
RTS

CALCHK  EQU *
CLR CHECK1
LDX #08
CALCHK1  RORA
BCC CALCHK2
INC CHECK1

CALCHK2  DECX
BNE CALCHK1
ROR CHECK1
RTS

PAKCHK - Calculates the address of the the 1st byte location
for the check data C1 to C5 and then rotates the RAM registers
CHECK0 and CHECK1 so that C1 to C5 are in the correct bit positions
for programming into the Check EEPROM.

Enter with C1 to C5 precalculated and stored CHECK0 (0-4), Start
address of DATA EEPROM in EPSTHI and EPSTLO and address offset
for data byte in RAM reg. INDEX.

Returns with C1 to C5 rotated into correct position within CHECK0
and CHECK1 (all other bit positions = 0) and address offset for
C1 to C5 in X-reg.
0408 1209 be 93  PAKCHK  LDX  INDEX
0409 120b a6 05  LDA  #$05
0410 120d 42  MUL  Multiply DATA offset by 5
0411  
0412  
0413 120e 3f 98  CLR  REM  Initialise remainder RAM reg.
0414  
0415  
0416  
0417  
0418 1210 56  RORX
0419 1211 46  RORA
0420 1212 24 02  BCC  CHKAD1
0421 1214 10 98  BSET  0,REM
0422 1216 56  CHKAD1  RORX
0423 1217 46  RORA
0424 1218 24 02  BCC  CHKAD2
0425 121a 12 98  BSET  1,REM
0426 121c 56  CHKAD2  RORX
0427 121d 46  RORA
0428 121e 24 02  BCC  CHKAD3
0429 1220 14 98  BSET  2,REM
0430  
0431  
0432  
0433 1222 3f 95  CHKAD3  CLR  CHECK1
0434 1224 3d 98  TST  REM
0435 1226 27 09  BEQ  NOREM
0436 1228 98  CHKAD4  CLC
0437 1229 39 94  ROL  CHECK0  Rotate CHECK0 and CHECK1 left
0438  
0439  
0440 122b 39 95  ROL  CHECK1  - the no. of left shifts is equal
to the value held in REM
0441 122d 3a 98  DEC  REM
0442 122f 26 f7  BNE  CHKAD4
0443  
0444 1231 97  NOREM  TAX  Store CHECK EEPROM offset for C1 to C5
in X reg.
0445  
0446 1232 81  RTS
0447  
0448  
0449  
0450  
0451  
0452  
0453  
0454  
0455  
0456  
0457 1233 cd 12 4f  EPRWRT  JSR  EXSTAI  Load ext sta inst subroutine in RAM
0458 1236 1e 09  BSET  WE,PCR  Select write mode
0459 1238 20 05  BRA  EWRT1
0460  
0461 123a cd 12 4f  EPRERA  JSR  EXSTAI  Load ext sta inst subroutine into RAM
0462 123d 1f 09  BCLR  WE,PCR  Select erase mode
0463 123f 11 09  BWRT1  BCLR  PLE,PCR  Latch address
0464 1241 bd 80  JSR  ADSTA  write to EEPROM
0465 1243 13 09  BCLR  PGE,PCR  Activate charge pump
0466 1245 cd 12 60  JSR  EPRDEL  Delay 10 ms at 2.5 MHz internal bus speed
0467 1248 10 09  BSET  PLE,PCR  
0468 124a be 91  LDX  SAVX  Restore X reg contents
0469 124c b6 90  LDA  SAVA  Restore acc. contents
0470 124e 81  RTS
0471  
0472 124f b7 90  EXSTAI  STA  SAVA  Save acc
0473 1251 bf 91  STX  SAVX  Save X reg
0474 1253 a6 d7  LDA  #$D7  Indexed 2 byte offset STA inst
0475 1255 b7 80  STA  ADSTA
0476 1257 a6 81  LDA  #$81
0477 1259 b7 83  STA  ADSTA+3
0478 125b b6 90  LDA  SAVA
0479 125d be 91  LDX  SAVX
0480 125f 81  RTS  
0481  
0482 1260 a6 d8  EPRDEL LDA  #$D8  10 ms delay at 2.5 MHz internal bus freq.
0483 1262 ae 10  EDEL1 LDX  #$10
0484 1264 cd 12 6b  JSR  EDEL2
0485 1267 4a  DECA
0486 1268 26 f8  BNE  EDEL1
0487 126a 81  RTS  
0488  
0489 126b 5a  EDEL2  DECX
0490 126c 26 fd  BNE  EDEL2
0491 126e 81  RTS  
0492  
0493  
0494  

116
**INTRODUCTION**

The MC68HC05B serial/parallel programmer module (Figure 1) enables the user to program MC68HC805B6 and MC68HC705B5 MCU devices. This application note describes the various operating modes of the module and gives details of its use and construction.

**PROGRAMMING MODES**

Two programming modes are available via jumper selection: parallel mode and serial mode.

In parallel programming mode, the user program contained in an external EPROM is copied into the internal EEPROM or EPROM of the MCU device, whereas in the serial programming mode the MCU EEPROM or EPROM can be programmed or read via the serial port on the programmer module.

Note: If the security bit is active on the 68HC705B5 device, no programming operations are possible. On the 68HC805B6 device, the device will be initially erased if programming is attempted with the security bit active.

Table 1 describes the 4 modes of operation for the 68HC805B6 and 68HC705B5 devices. The markings for jumpers J2 and J3 are on the programmer board.

<table>
<thead>
<tr>
<th>Jumper J2</th>
<th>Jumper J3</th>
<th>Device 68HC805B6</th>
<th>Device 68HC705B5</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIAL</td>
<td>SERIAL</td>
<td>SERIAL LOAD (NO ERASE)</td>
<td>RAM/EPROM SERIAL BOOT</td>
</tr>
<tr>
<td></td>
<td>PARALLEL</td>
<td>SERIAL LOAD WITH ERASE</td>
<td>EPROM ERASE CHECK</td>
</tr>
<tr>
<td></td>
<td>PARALLEL</td>
<td>PARALLEL RAM BOOT</td>
<td>PARALLEL RAM BOOT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PARALLEL EEPROM BOOT</td>
<td>PARALLEL EEPROM BOOT</td>
</tr>
</tbody>
</table>

Table 1. Programming modes for 68HC805B6 and 68HC705B5 devices

**PARALLEL PROGRAMMING MODE**

This mode enables programming of the MCU EEPROM (68HC805B6) or EPROM (68HC705B5) directly from an external 27C64 EPROM device, with the programming module operating as a stand-alone unit. The main functions of erasure (68HC805B6 only), programming, and verification are all implemented in this mode.

In this mode, all the internal EEPROM of the 68HC805B6 is automatically erased before being programmed. Any failure of the EEPROM to erase results in illumination of the red LED and a re-attempt of erasure. EEPROM erasure is normally complete within 500 mS. The erased state of the 68HC805B6 is $FF and $00 for the 68HC705B5.

The 27C64 EPROM should contain the data to be programmed with the same addresses as the 6 Kbyte EEPROM of the 68HC805B6 and so the EPROM should only have data between addresses $000 and $1FFF inclusive. Note: The smaller 256 byte EEPROM array of the 68HC805B6 is not programmable from external EPROM.

When programming 68HC805B6 devices, locations of the external EPROM not in the internal EEPROM address range are omitted, as are locations containing the data $FF, thus speeding up the programming operation. With 68HC705B5 devices, a similar technique is used, with the exception that areas with the data $00 are omitted.
Figure 1. MC68HC805B6 and MC68HC705B5 Serial/Parallel Bootstrap Programmer

PARTS LIST

<table>
<thead>
<tr>
<th>RESISTORS</th>
<th>CAPACITORS</th>
<th>INTEGRATED CIRCUITS AND SOCKETS</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 3K</td>
<td>C1-C4 22μF</td>
<td>IC1 68HC805B6 52 Pin PLCC ZIF</td>
</tr>
<tr>
<td>R2 3K</td>
<td>C5 0.01μF</td>
<td>IC2 INT27C64 28 Pin DIL ZIF</td>
</tr>
<tr>
<td>R3-R5 100K</td>
<td>C6 1.0μF</td>
<td>IC3 MAX232CPE 16 Pin DIL LIF</td>
</tr>
<tr>
<td>R6 1K</td>
<td>C7,C8 22pF</td>
<td></td>
</tr>
<tr>
<td>R7,R8 470</td>
<td>C9 100μF</td>
<td></td>
</tr>
<tr>
<td>R9 10M</td>
<td>C10,C11 47μF</td>
<td></td>
</tr>
<tr>
<td>R10 4K7</td>
<td>C12 1.0μF</td>
<td></td>
</tr>
<tr>
<td>R11 10K</td>
<td>C13 0.1μF</td>
<td></td>
</tr>
<tr>
<td>R12,R13 1K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14 4K7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15-R37 100K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R39 10K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R40 1K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R41 12K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R42 4K7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15-R37 100K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R39 10K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R40 1K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R41 12K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R42 4K7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DIODES

| D1,D2 | 1N914 |
| D3 | LR3160 |
| D4 | LG3160 |
| D5,D6 | 1N5822 |
| D7 | 1N914 |
| D8 | 1N5818 |

SWITCHES

| S1,S2,S3 | 2 Way, Toggle Switch (SPDT) |

MISC

| CR1 | MK04000A 4MHz Crystal Package HC-18U |

TRANSISTORS (All in TO92 Package)

| Q1,Q2 | BC337-25 |
| Q3 | BC239C |
| Q4 | BC309C |
During the programming operation the green LED should flash with a period of approximately 1 second to indicate normal programming mode.

After the programming operation has been completed, the programmed contents of the MCU are verified against the external EPROM. Any failure to verify will result in illumination of the red LED. Successful verification will result in illumination of the green LED.

The 68HC705B5 device can be checked for the EPROM in the erased state by placing the jumpers in the SERIAL and ERASE + BOOT positions with +5 volts on the Vpp supply for 68HC705B5. In this case follow the instructions below for parallel programming ignoring steps 4 and 5, but there is no need for a 27C64 EPROM in socket IC2. The green LED turned on indicates success, the red LED indicates that the EPROM is not in the erased state.

**PARALLEL PROGRAMMING OPERATION**

To program the MCU from an EPROM using parallel mode, perform the following steps:

1. With power to the module removed install MCU and EPROM devices into the programming module.
2. With the power switches S1 and S2 both off, and switch S3 in the RESET position, connect both the +5V supply and appropriate Vpp supply (68HC705B5 or 68HC805B6) to the module.
3. Set jumper J4 to the appropriate setting for the MCU being programmed (705B5 or 805B6).
4. Set jumper J3 to the ‘ERASE + BOOT’ position.
5. Set jumper J2 to the ‘PARALLEL’ position.
6. Turn the +5V power supply switch, S1, ON.
7. Turn the Vpp power supply switch, S2, ON.
8. Place switch S3 in the RUN position.
9. Once the green LED has stopped flashing, and remains continuously illuminated, place switch S3 to the RESET position.
10. Place the Vpp power supply switch, S2, in the OFF position.
11. Place the +5V power supply switch, S1, in the OFF position.

Note: To avoid possible damage to the MCU it is essential that power to the programming module is applied and removed in the sequence specified above.

**SERIAL PROGRAMMING MODE**

This mode allows the user to program and read the MCU EEPROM or EPROM via the serial port on the programmer module. By using a host computer and a control program such as E2B6, data can be downloaded and programmed onto the MCU, or uploaded from the MCU back to the host computer.

Programming in serial mode consists of the MCU reading a byte of data from the serial port, programming it into the internal 6 Kbyte EEPROM or EPROM array, reading the data back from programmed location and sending it to the serial port. The host computer should verify programming by checking the data returned from the programming module for differences from the programmed data, which would indicate incorrect programming or erasure.

As in parallel mode, bytes to be programmed with $FF for EEPROM devices and $00 for EPROM devices are skipped, reducing the overall programming time and allowing the memory upload feature to be implemented. This involves the host computer reading the data programmed in the MCU by attempting to program these values and examining the returned verification data.

**MC68HC805B6**

A program called E2B6 is available for the IBM PC and similar machines that communicate via RS-232 with the programmer board serial connector. This program allows upload (data transfer from 68HC805B6 to IBM PC) to read the EEPROM and can also program the EEPROM by downloading S1 record files to the 68HC805B6 device.

As in the parallel programming mode, the internal EEPROM areas can be automatically erased before programming. In serial mode, however, this operation is optional, and is selected by setting jumper J3 to the ERASE + BOOT position. An exception is if the EEPROM security bit is active, in which case the erase will be carried out regardless of the setting on J3. A ‘read’ or ‘upload’ of the EEPROM will also cause the EEPROM to be erased if J3 is set to ‘ERASE + BOOT’ or if the security bit is active.

**MC68HC05B6**

The 68HC05B6 (ROM device) 256 byte EEPROM may also be programmed using this board as described in
application note AN434. In this case the jumpers should be set as for the 68HC805B6, ERASE+BOOT and SERIAL but the Vpp supply for the 805B6 power socket should be connected to +5 volts.

**MC68HC705B5**

A program called EPB5 for the IBM PC communicates via RS-232 with the programmer board serial connector. This program allows upload (data transfer from 68HC705B5 to IBM PC) to read the EPROM and can also program the EPROM by downloading S1 record files to the 68HC705B5 device.

The 68HC705B5 EPROM means that the jumpers J2 and J3 have slightly different meaning. See table 1 for details of operating modes.

**SERIAL PROGRAMMING OPERATION**

1. Run the program E2B6 (68HC805B6) or EPB5 (68HC705B5) on an IBM PC to communicate with the device to be programmed.
2. With power to the module removed install the MCU and connect the serial line between the host computer and the serial port on the module.
3. With the power switches S1 and S2 both off and switch S3 in the RESET position, connect both the +5V supply and appropriate Vpp supply (705B5 or 805B6) to the module.
4. Set jumper J4 to the appropriate setting for the MCU being programmed (705B5 or 805B6).
5. Jumper J3 should be set to the desired setting, e.g., BOOT ONLY if reading data from the MCU or ERASE + BOOT if re-programming a device (68HC805B6 only).
6. Set jumper J2 to the 'SERIAL' position.
7. Turn the +5V power supply switch, S1, ON.
8. Turn the Vpp power supply switch, S2, ON.
9. Place switch S3 in the RUN position when prompted by the host computer control program.
10. Follow the instructions of the upload/download program to initiate the data transfer.
11. When the operation has been completed, place switch S3 to the RESET position.
12. Place the Vpp power supply switch, S2, in the OFF position.
13. Place the +5V power supply switch, S1, in the OFF position.

For programming several devices, leave the IBM PC program running and repeat instructions 2 to 13 inclusive.

Note: The documentation of the host computer control program being used should be consulted for further details on the use of serial programming mode.
INTRODUCTION

The MC68HC05E0 is a versatile member of the M6805 family of microprocessors. Unlike most other versions it has no on-chip ROM but instead can address a full 64K of external memory. This memory could simply be a ROM or EPROM containing the required program but can also include RAM and/or additional hardware. In addition to the external busses required to support this capability, the MC68HC05E0 has the usual I/O, timers etc. found on single-chip microprocessors.

The EPROM emulator described here illustrates a typical application of this type of microprocessor. In addition to the program EPROM it employs a keyboard, LCD, serial communication and 64K of paged RAM. The emulator can replace with RAM the program EPROM or ROM (up to 64K x 8) in a microprocessor based target system. This is done by connecting the emulator to the target system via a cable to its EPROM socket.

The object code, which can be loaded serially or from an EPROM, can be inspected and modified with the use of a local keyboard and LCD display. The new or modified code can then be used by the target system without having to go through the procedure of erasing and re-programming an EPROM after each software change. A selectable offset in $0100 steps is available in order to position the code correctly in the target system’s memory map.

The emulator facilitates the debugging of hardware and software for any system whose control program is to be contained in a 27(C)16/32/64/128/256/512 type EPROM. The control software includes branch offset calculation for 6805 code and is thus particularly suitable for debugging systems using one of the microprocessors from the M6805/01/11 ranges.

Two basic methods of loading a program are available. The first is applicable when the code is available in an existing EPROM. The contents of this EPROM can be transferred by the microprocessor into the RAM. This method requires an existing EPROM but will prove useful in applications where a small change to an existing program has to be checked before committing to an updated EPROM. This can be done without access to the source or object code. An EPROM can be read from the target system interface (through the emulator’s buffers) or from a separate socket wired directly to the microprocessor. The former method allows one socket to be used for both EPROM reading and the target cable. The second method saves having to remove the target cable to read an EPROM but requires an additional socket.

Alternatively, data can be serially loaded in the form of Motorola S-records via an RS232 link. This code can come from a “COM” port of a PC (using the COPY command) or by tapping into the link between a computer and its terminal on a system using an RS232 connection between terminal and host. In this case a TYPE or LIST to the terminal should be used. A verify facility which compares the contents of RAM with serial S-records is also available, as is a routine to dump the current contents of the emulation RAM out on the RS232 interface.
There are three modes of operation:

a) Mode A allows the microprocessor to read the contents of an EPROM on the target system interface (this is most easily arranged by connecting the cable to the target system via a zero or low insertion force socket) by enabling buffers B1 and B2 to drive from left to right to supply addresses to the socket (the RAM also receives these addresses but its data outputs are disabled). Buffers B3 and B4 are enabled from right to left to return the data from the EPROM to the MC68HC05E0. The RAM is disabled via its chip-enable pin and so does not affect the data bus between buffers B3 and B4.

b) Mode B enables the buffers in such a way that the microprocessor can read from and write to the RAM. B1 is enabled to supply addresses to the RAM from the microprocessor (MC74HC245s were used throughout although a bidirectional buffer is not strictly necessary in this position as B1, if enabled, always drives from left to right). B3 is enabled to allow data to be written to or read from the RAM. The direction control for B3 is by the R/W signal from the MC68HC05E0 (gated with the RAM’s chip enable). This mode is used during use of the memory modify facilities. Buffer B4 is disabled so that there is not a bus contention on either of its busses even if the target system is still connected. Buffer B2 is also disabled. The routine (L1) which loads RAM from an EPROM on the target system interface switches between modes A and B for each byte transferred.

c) In the emulation mode (C) the target system plugged into the socket is required to have access to the RAM so buffers B2 and B4 are enabled. B2 passes the addresses from right to left and B4 the data from left to right (as the emulation is for an EPROM, the target system is not allowed to write to the RAM). Buffers B1 and B3 are disabled.
CIRCUIT

Figure 2 shows the main circuit. An MC74HC138 is used to provide the chip enables. The emulator hardware is enabled in the address range $4000 – $7FFF and the EM64K program EPROM (27C164) at $C000 – $FFFF. If the EM64K program is contained in a 27C16 its pin 21 (Vpp) should be held high.

An additional socket is shown at address $8000 – $BFFF. This is for the optional LOAD2 facility which allows code to be loaded from EPROM without having to disconnect the cable to the target system. The emulation RAM occupies the address range $4000 – $7FFF. As this is only a 16K address space the RAM is paged. The four pages are selected by I/O lines (port B, bits 0 and 1) from the microprocessor. The memory map of the emulator is shown in figure 7.

The control lines (port B, bits 4–7) are biased by resistors. This holds the system in mode B if the MC68HC05E0 is held in reset and prevents bus contention resulting from an illegal combination of control signals. During hardware debug of the emulator it is advisable to use a current limited power supply (in the range 50–100 mA) as a bus contention can cause sufficiently high currents to damage the buffers.

The display is a 6-digit 4-backplane LCD (eg Hamlin type 4200 or the 8-digit GE type LXD69D3F09KG) which is driven by an MC145000 display driver. The driver is controlled by a 2-line serial link from the microprocessor. A single-backplane (or “static”) display can be used as an alternative as shown in Figure 3. Three MC144115 driver chips are used. This circuit requires many more connections to the LCD but allows the use of a more readily available display. A third line (port B bit 3) from the microprocessor is used to supply the enable pins of the MC144115s. The single-backplane display drivers can be supplied directly from the main 5 volt supply but the multiplexed display requires a lower voltage. Figure 2 shows the MC145000 supplied via a 20k potentiometer which serves as a contrast control.

The keyboard uses an MC14028 decoder to minimise the number of I/O pins used. Note that port A bit 6 is used for both the keyboard and the display driver. The LOAD1 and LOAD2 keys overwrite the contents of emulation RAM and should thus not be pressed accidentally. It may therefore be useful for only those LOAD keys actually required to be fitted (usually LOAD1 and LOAD2 will not both be required) and for any parallel LOAD keys fitted to be placed away from the front panel or protected by requiring two keys, connected in series, to be pressed. An accidental press of the serial LOAD key can be aborted by pressing RESET.

As the circuit, except for the RS232 interface, is all CMOS the supply current is very low when the microprocessor is in STOP mode. This is a low power mode in which all processing, even the clock, is stopped. In the emulation mode (C) the MC68HC05E0 is in stop mode. In this mode and with no bus activity from the target system (or its interface open circuit) the supply current should be less than 1 μA (this does not include the current taken by the RS232 interface which, if present, can be switched off when not in use, or the 70–80 μA taken by the LCD driver).

It is worth checking that a low supply current is achieved as any excess can be a useful pointer to a wiring fault, particularly open circuit pins. The supply current may be affected by the choice of RAM but the MCM60L256 selected has a specified standby ICC of 2 μA and is typically well below this figure. In many applications the full 64K of RAM will not be required. If this is the case, only the required RAM need be included. 6116 2K RAMs could be used for 2–4K applications and MCM60L64s or equivalents for 8–16K. If using 6064s, their second chip enable pin (E2) should be held high. One MCM60L256 provides 32K. The serial load routine includes a read-back check on each byte sent to RAM so an attempt to write to non-existent RAM will generate an error message indicating the first faulty address. If 16K or less is required then the two 74HC245s handling addresses, A14 and A15, can also be omitted. These buffers have unused pins. The simplest way to ensure that no pins are left open circuit is to wire up the buffers in a manner similar to those actually used. Pins 2–7 of the left-hand buffers are held high while pins 13–18 are connected to the right-hand buffers whose other pins have pull-ups. This arrangement means that there will be no open circuits or bus contentions regardless of the levels of the control lines. If only one memory chip is used, the 74HC00 can be omitted (connect pin 3 of the 74HC32 directly to the RAM’s chip enable).

The optional RS232 interface can most easily be implemented using the single-supply MC145407 driver-receiver chip. If outputting of S-records is not required then a simple transistor inverter with a pull-up resistor and a reverse polarity protection diode can be used. This interface is shown in Figure 4.
Figure 2. EPROM emulator circuit diagram

6 x 4 Key Matrix

Esc EM M L2
L1 Load Verif. ENT.
7 8 9 A (Mode)
4 5 6 B (Cursor)
1 2 3 C (Dump)
0 F E D

MC145000P

MC68 HC05 E0FN

Optional Socket (Socket2)

MC145407P

MC74 HC138

27C64 (EM64K)

Target System EPROM Interface

MCM68L256

Addresses

2 x

MC144049

MC14028P

Resistors

Capacitors

10µF

100nF

8.2µF

4 x 10k

250µF

8 x 10k

10k

50k

1k

68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
Figure 3. Alternative static LCD display

Figure 4. Simple input-only RS232 interface
**IDD Monitor**

It is often useful with CMOS circuits to provide a simple $I_{DD}$ monitor which shows via an LED whether or not the $I_{DD}$ is above or below a set value. In this application it shows whether or not the microprocessor is in the STOP mode. The required circuit is shown in Figure 5. The current threshold can be chosen by selecting the value of $R_1$. A value of $1\,k\Omega$ sets the limit at about 500 $\mu$A which means the LED should be off in the emulation mode but on otherwise. The 500 $\mu$A limit allows the LCD and perhaps an emulator-supplied CMOS target system to be supplied without switching on the LED. When the microprocessor is not in STOP (emulator not in mode C), its $I_{DD}$ is several milliamps and the LED should be lit. In a battery application this circuit would also serve as a useful reminder that the RS232 interface has been left on. If a multiplexed LCD is used it may be preferable not to supply it via this type of monitor circuit as a significant change in contrast may occur when the microprocessor goes into its STOP mode (see Figure 5). The monitor drops about 600 mV when the microprocessor is running so the supply voltage should be chosen accordingly; four zinc-carbon or five Ni-Cad cells were found to be satisfactory.

**Address trap**

The emulator allows memory locations to be examined and changed, but does not provide the breakpoint and trace features normally found in development systems. A limited capability can be made available if address comparators of the type shown in Figure 6 are added. This circuit gives an LED indication if the address selected on the bank of switches is encountered by the program running in the target system. An address coincidence is latched by the 74HC74. To indicate the occurrence of a repetitive event, a one-shot chip could be added.

**SERIAL LOAD**

To load external Motorola S-records the serial load key (LOAD) should be pressed. The LCD will display "LOAd". S-records should then be supplied at 9600 baud (8-bit, no parity) on the RS232 interface. When an S9 termination record is received, the prompt returns. If an error is detected during a serial load, the load routine stops and displays the address at which the error occurred and the error type.

![Figure 5. Simple IDD monitor](image)
The following error types are possible:

1: Checksum error, transmitted data or interface faulty.
2: RAM read-back error, RAM faulty or non-existent.
3: ASCII character less than $30 (0) received.
4: ASCII character between $39 (9) and $41 (A) received.
5: ASCII character more than $46 (F) received.
7: Verify error when comparing S-records with emulation RAM.

If, when using the emulator, the target system ceases to function properly, then the verify function can be used to check that the emulation RAM has not been corrupted. The VERIFY function is used exactly like LOAD except that RAM is compared with, rather than loaded by, the S-records.

The address at the start of each S1-record determines the address at which the code will reside in the target system. This address will sometimes be different from that at which the code is required to be loaded into the emulation RAM so an offset may need to be used. The offset byte is entered using the appropriate key and allows an offset of any multiple of $0100. The offset is subtracted from the MSB of the S-record address and this modified address is the physical address at which the data is loaded into the emulation RAM. The S-record output routine adds the offset before transmitting the records. At reset or power-up the offset is initialised to zero.

All addresses entered while using the MEMORY-MODIFY, BRAOFF and DUMP routine use the actual address in the target system. These addresses will only be the same as the physical RAM address if the offset is zero.

Figure 6. Address Trap
PARALLEL LOAD

When the parallel load functions (LOAD1 and LOAD2) are used, OFFSET has no effect on the transfer of code into RAM. It can, however, still be used to offset the RAM addresses to correspond with the actual address in the target system program when using the memory-modify facilities.

27(C)64/128/256 EMULATION

When emulating a 27(C)512 EPROM, all the RAM is used. For emulation of smaller EPROMS, less RAM is required. The memory used will be at the beginning of RAM (starting at address zero) only if the unused high-order address lines are held low. It may, however, be more convenient to allow one or more of these addresses to be high. The pull-ups included in Figure 2 will hold any uncommitted lines high. For example, a 27(C)64 can be emulated with no hardware change as long as the code is loaded between $E000 and $FFFF. This will often be appropriate as it allows the vectors at the top of the target system’s memory to be included. It makes little difference if the target microprocessor has an address space smaller than 64K as the high order addresses will not be present and will be held high. Clearly, the code must still be assembled at the appropriate addresses and the emulator’s offset feature used to load the S-records between $E000 and $FFFF. Alternatively, the S-records can be loaded lower in the emulator’s address space and the relevant high-order addresses held low. When loading from a smaller EPROM, with a VPP or PRG pin, these pins should be configured correctly for reading (high) and not driven by the emulator. See Figure 2 for the industry-standard EPROM pinouts. As the software does not behave differently for smaller EPROMs, a full 64K transfer will still be made, copying the EPROM several times into the 64K RAM. The actual copy used depends on the levels of the high order addresses as outlined above.

TARGET SYSTEM INTERFACE

Vdd can be connected to the interface by the link shown. The simplest method of use is to make this connection and to use a common supply for the emulator and the target system. If, however, separate supplies are used, then pin 28 should not be connected. If separate supplies are used, care should be taken that they do not differ by more than 0.5V. A delta greater than this may cause a malfunction as a result of the logic level on an input pin being in excess of the chip’s Vdd.

In emulation mode the target system has total control of the RAM except for its RW line. It can thus use the RAM exactly as if it were a ROM or EPROM. Before IRQ (or RESET) is pressed to exit from the emulation mode the target system should be stopped so that it no longer expects the “EPROM” to be there. This will normally be done by holding the target system in reset. If the target system is an M68(HC)05 (eg MC68HC05E0 or MC146805E2) or M68HC11, then it can alternatively be put into its STOP mode. If this is its normal idle condition, then nothing need be done prior to exiting emulation.

EM64K PROGRAM

The EM64K control program is less than 2K bytes long and can thus reside in a 27C64 or 27C16. The circuit is shown for a 27C64 and assumes that the program starts at the beginning of the EPROM. This EPROM is enabled at $C000 (and $E000 as A13 is not used). An assembled listing of the control program is included at the end of this application note.
## EM64K KEY FUNCTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>KEY</th>
<th>Description of function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD1</td>
<td>L1</td>
<td>Load RAM from target system interface ($4000–$7FFF).</td>
</tr>
<tr>
<td>LOAD2</td>
<td>L2</td>
<td>Load RAM from secondary socket ($8000–$BFFF).</td>
</tr>
<tr>
<td>SERIAL LOAD</td>
<td>LOAD</td>
<td>Load emulation RAM with S-records via the RS232 interface, during loading LCD shows “LOAd”.</td>
</tr>
<tr>
<td>VERIFY</td>
<td>Verf</td>
<td>Compare emulation RAM with S-records via the RS232 interface, LCD displays “UErIFy”.</td>
</tr>
<tr>
<td>EMULATE</td>
<td>EM</td>
<td>Emulator mode. Prompts: “EP ?” for the removal of an EPROM (if present) and connection of the target system (press again if OK) and put micro into EMULATE mode.</td>
</tr>
<tr>
<td>MEMORY MODIFY</td>
<td>M</td>
<td>Display/change a RAM location. When pressed the last address is displayed. Press ENTER to display the contents of this address or input a new address followed by ENTER. To change, input new data followed by ENTER. ENTER moves to next address, M moves to previous address, ESCAPE exits.</td>
</tr>
<tr>
<td>ENTER</td>
<td>Ent</td>
<td>Enter keyed-in address or data (and move to next address in MEMORY MODIFY).</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>Esc</td>
<td>Exit from current function (OFFSET, BRAOFF, DUMP or MEMORY MODIFY).</td>
</tr>
<tr>
<td>BRAOFF</td>
<td>A</td>
<td>Calculate branch offset. The address of the branch instruction and of the destination are requested. If a valid branch is calculated it is written into memory and displayed. If not valid then “or” for out of range is displayed. A branch of -128 through +127 relative to the start address of the next instruction is allowed. Esc returns to the normal prompt.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>B</td>
<td>Allows entry of an offset to the emulation RAM address. It is subtracted from the most significant byte of the address specified by the incoming S-records. The offset is added to the address by the DUMP function.</td>
</tr>
<tr>
<td>DUMP</td>
<td>C</td>
<td>Output emulation RAM contents as S-records via RS232 interface. RAM start and finish addresses are requested. They should be entered followed by ENTER. After the second ENTER, the S-record output starts.</td>
</tr>
<tr>
<td>IRQ</td>
<td>IRQ</td>
<td>Abort emulation and return to emulator monitor.</td>
</tr>
<tr>
<td>RESET</td>
<td>RESET</td>
<td>Resets emulator, displays prompt (□). Should be used after power-up or if the emulator malfunctions. Can be used instead of IRQ, with the difference that the OFFSET is reset to zero. RESET provides the only exit from a LOAD or VERIFY which has not been terminated correctly by the reception of an S9 record.</td>
</tr>
</tbody>
</table>
SOFTWARE

A listing of the control program used in the emulator is included in this application note. Some points specific to the MC68HC05E0 are discussed below.

Port D on the MC68HC05E0 can be used as a normal I/O port or can selectively supply special signals. In this application five of the special function are used. These function are selected using the register at address $12. The function used are P02, RW, A13, A14 and A15. By default only addresses A0 through A12 are available as this will be sufficient in many applications. In this application, however, all the addresses are required. The clock (P02) is used to qualify the chip selects generated by the MC74HC138 and R/W for control of the emulation RAM. The other three pins are left as I/O pins but are not used in this application. The initialisation of $12 can be seen on lines 93 and 94 of the software listing.

The only other register used (apart from the I/O data and DDR registers) is the interrupt control register ($0E). It is written to $01 on lines 82 and 83 of the listing. This operation clears the interrupt flag (bit 3) but keeps the INTMX bit set. This bit enables external interrupts. The registers associated with unused on-chip resources are left at their reset conditions. An important bit in the MC68HC05E0 is the XROM bit ($2,$0C). It defaults to a 1 which is appropriate in this application. When it is cleared it constrains the data bus to be input only thus preventing any unnecessary activity in sensitive applications when writing to external memory is not required.

The MC68HC05E0 has the 8-bit index register common to all M6805 microprocessors. It is thus not able to contain a 16-bit extended address. For this reason, loading and storing in the emulator's RAM is carried out using a small program in the micro's RAM. This program consists of an extended LDA or STA instruction followed by a two byte address which can be built in software and an RTS instruction. The four-byte program resides in RAM at locations W2, ADDLEH, ADDRL and W3. It allows the full 64K map to be accessed using addresses generated within the program. Address generation is further complicated by the requirement that the emulation RAM is in four 16K pages. The two most significant addresses thus have to be transferred to port lines PB0 and PB1.

SERIAL INTERFACE

Figure 8 shows a suggested method of wiring up the RS232 sockets in an emulator with both loading and dumping capabilities. This arrangement facilitates use of the serial LOAD and DUMP routines of the emulator either via a PC COM port or between a host and terminal connected by an RS232 link. When using a PC the "host" socket should be used. As only one pin on the MC68HC05E0 is used, switching is required to make the required connections. S2 can be eliminated (or left at "L") if only loading is required, as will often be the case. To save power in battery applications, the RS232 interface chip can be switched off using S1. The following table shows possible methods of use.
<table>
<thead>
<tr>
<th>Set-up</th>
<th>Function</th>
<th>S1</th>
<th>S2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host &amp; terminal</td>
<td>Load</td>
<td>On</td>
<td>L</td>
<td>Terminal and host connected. Micro looks at data sent from host to terminal (pins 3).</td>
</tr>
<tr>
<td></td>
<td>Dump</td>
<td>On</td>
<td>D</td>
<td>Connection between terminal and host broken. S-records sent to both host (2) and terminal (3).</td>
</tr>
<tr>
<td>PC &quot;COM&quot; port</td>
<td>Load</td>
<td>On</td>
<td>L</td>
<td>S-records loaded from pin 3.</td>
</tr>
<tr>
<td></td>
<td>Dump</td>
<td>On</td>
<td>D</td>
<td>S-records sent to pin 2 on &quot;host&quot; socket (and pin 3 on &quot;terminal&quot; socket).</td>
</tr>
</tbody>
</table>

Figure 8. RS232 circuit with LOAD/DUMP switching
<table>
<thead>
<tr>
<th>Pin</th>
<th>27512</th>
<th>27256</th>
<th>27128</th>
<th>2764</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A15</td>
<td>Vpp</td>
<td>Vpp</td>
<td>Vpp</td>
</tr>
<tr>
<td>2</td>
<td>A12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>A5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>A4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>A3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>A2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>D0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>D1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>D2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Vss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>D3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>D4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>D5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>D6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>D7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Chip enable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>A10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Output enable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>A11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>A9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>A8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>A13</td>
<td>A13</td>
<td>A13</td>
<td>NC</td>
</tr>
<tr>
<td>27</td>
<td>A14</td>
<td>A14</td>
<td>PGM</td>
<td>PGM</td>
</tr>
<tr>
<td>28</td>
<td>Vcc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. 27(C) 512, 256, 128 and 64 pin-outs
(Table shows the standard 28-pin EPROMs.
Blank entries indicate that the pin is the same as for the 27(C) 512.)
ASSEMBLED LISTING OF THE EM64K CONTROL PROGRAM

---

MC68HC05ED EPROM Emulator.

A 68HC05ED is used to emulate an EPROM of up to 64K (27C512) with SRAM which can be loaded from an EPROM or serially by S-records via an RS232 interface and changed if required for de-bug etc.

P. Topping 11-Jan-91

---

PORTA EQU $00  PORT A ADDRESS
PORTB EQU $01  " B "
PORTC EQU $02  " C "
PORTD EQU $03  " D "
PORTE EQU $04  " E "
PORTAD EQU $05  PORT A DATA DIRECTION REG.
PORTBD EQU $06  " B "
PORTCD EQU $07  " C "
PORTDD EQU $08  " D "
PORTED EQU $09  " E "
ICR EQU $0E  INTERRUPT CONTROL REGISTER
PORTDSF EQU $12  PORT D ALTERNATIVE FUNCTION REGISTER

ORG $0020  RAM ALLOCATION

DTABL RMB 6  LCD BUFFER
TEMP RMB 2
W1 RMB 1
W2 RMB 1  RAM SUBROUTINE  LOA or STA
ADDEH RMB 1  " "
ADDR RMB 1  " "
W3 RMB 1  " LSB"
W4 RMB 1  " RTS"
W5 RMB 1
W6 RMB 1
ADDH RMB 1
STAT RMB 1  STATUS BYTE :
* 2: REAL ADDRESS (OFFSET)
* 4: INVALID ADDRESS (BLDRNG)
* 5: VERIFYING (LOAD)
* 6: INDIVIDUAL REG. (MEMEX)
* 7: PUNCH END

CHSUM RMB 1  CHECKSUM
COUNT RMB 1  BIT COUNTER
TMP1 RMB 1
TMP2 RMB 1
BCNT RMB 1  S-RECORD BYTE COUNT
ERTYP RMB 1  ERROR TYPE
ERDAT RMB 1  ERROR DATA
OFF RMB 1  S-RECORD OFFSET
RMB 1  UNUSED
STACK RMB 12  13 BYTES USED (1 INTERRUPT
SP RMB 1  AND 4 NESTED SUBROUTINES)

---

em64k.as5

133
Idle loop and routine to decide which key has been pressed.

**ORG $E000**

```assembly
68 0000e000 cde05f SCAN JSR KEYSNC KEY FOUND?
69 0000e003 24fb BCC SCAN NO, TRY AGAIN
70 0000e005 5f CLRX
71 0000e006 b72b STA W1 CODE OF PRESSRED KEY
72 0000e008 d6e09f RJ LDA CTAB.X FETCH KEYCODE
73 0000e00b b128 CMP W1 THIS ONE?
74 0000e00d 270b BEQ PJ YES
75 0000e00f c1e0bf CMP LAST NO. LAST CHANCE?
76 0000e012 273a BEQ GETCMD YES, ABORT
77 0000e014 5c INCX NO
78 0000e015 5c INCX TRY
79 0000e016 5c INCX THE
80 0000e017 5c INCX NEXT
81 0000e018 20ee BRA RJ KEY
82 0000e01a 0001 PJ LDA #1
83 0000e01c b70e STA ICR CLEAR IRQX FLAG
84 0000e01e 5c INCX
85 0000e01f dce09f JMP CTAB.X
```

**START LDA #$E3 ENABLE PORTD SPECIAL FUNCTIONS**

```assembly
93 0000e022 a6e3 STA PORTSF P02, R/W, A13, A14 & A15
94 0000e024 b712 CLR PORTA
95 0000e026 3f00 LDA #$F0 DISPLAY/KEYBOARD
96 0000e028 a6f0 STA PORTAD I/O
97 0000e02a b705 LDA #$5B MODE 2, ENABLE (144115) HIGH
98 0000e02c a658 STA PORTB
99 100 0000e02e b701 LDA #$FB BITS 0, 1, 3-7 OUTPUTS
101 0000e030 a6fb STA PORT80 BIT 2 INPUT
102 0000e032 b706 LDA #$FF ALL OUT, NOT USED
103 0000e034 3f02 CLR PORTC
104 0000e036 a6ff LDA #$4F STA PORTCD
105 0000e038 b707 CLR PORTD
106 0000e03a 3f03 LDA #$7C BITS 2, 3 & 4, OUT, NOT USED
107 108 0000e03c b708 STA PORTOD
109 0000e03e a61c LDA #$4B BITS 0-3 OUT, NOT USED
110 0000e040 3f04 CLR PORTE
111 112 0000e042 a60f LDA #$0F STA PORTED
112 0000e044 b709 CLR STAT
113 0000e046 3f31 LDA #$1C INITIALISE
114 0000e048 3f2b CLR ADDR8
115 0000e04a 3f30 LDA瘆ADDR
116 117 0000e04c 3f39 CLR OFF
118 012 0000e04e a659 GETCMD LDA #$5B MODE 2, ENABLE (144115) HIGH
```

**RESETroutine.**

---

134
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0000e050 b701</td>
<td>STA PORTB</td>
</tr>
<tr>
<td>124</td>
<td>0000e052 cde235</td>
<td>JSR CLRTAB</td>
</tr>
<tr>
<td>125</td>
<td>0000e055 e633</td>
<td>LDA #$33</td>
</tr>
<tr>
<td>126</td>
<td>0000e057 e720</td>
<td>STA DTABL</td>
</tr>
<tr>
<td>127</td>
<td>0000e059 cdelec</td>
<td>JSR DISTAB</td>
</tr>
<tr>
<td>128</td>
<td>0000e05c 9c</td>
<td>RSP</td>
</tr>
<tr>
<td>129</td>
<td>0000e05d 20a1</td>
<td>BRA SCAN</td>
</tr>
</tbody>
</table>

em64k.as5

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>131</td>
<td>0000e05f 4f</td>
<td>KEYSCN CLRA</td>
</tr>
<tr>
<td>132</td>
<td>0000e060 ae06</td>
<td>LDX #6</td>
</tr>
<tr>
<td>133</td>
<td>0000e062 ab10</td>
<td>KEY1 ADD #$10</td>
</tr>
<tr>
<td>134</td>
<td>0000e064 b700</td>
<td>STA PORTA</td>
</tr>
<tr>
<td>135</td>
<td>0000e066 b600</td>
<td>COLUMN LDA PORTA</td>
</tr>
<tr>
<td>136</td>
<td>0000e068 c719</td>
<td>STA W3</td>
</tr>
<tr>
<td>137</td>
<td>0000e06a a50f</td>
<td>BIT #$0F</td>
</tr>
<tr>
<td>138</td>
<td>0000e06c b72c</td>
<td>BEQ COLRET</td>
</tr>
<tr>
<td>139</td>
<td>0000e06e 6d1d</td>
<td>BSR DBOUNC</td>
</tr>
<tr>
<td>140</td>
<td>0000e070 b600</td>
<td>LDA PORTA</td>
</tr>
<tr>
<td>141</td>
<td>0000e072 b12c</td>
<td>CMP W3</td>
</tr>
<tr>
<td>142</td>
<td>0000e074 b611</td>
<td>BNE COLRET</td>
</tr>
<tr>
<td>143</td>
<td>0000e076 99</td>
<td>COL1 LDA PORTA</td>
</tr>
<tr>
<td>144</td>
<td>0000e077 b600</td>
<td>BIT #$0F</td>
</tr>
<tr>
<td>145</td>
<td>0000e079 a50f</td>
<td>BNE COL1</td>
</tr>
<tr>
<td>146</td>
<td>0000e07b 26fa</td>
<td>BSR DBOUNC</td>
</tr>
<tr>
<td>147</td>
<td>0000e07d a0de</td>
<td>LDA PORTA</td>
</tr>
<tr>
<td>148</td>
<td>0000e07f b600</td>
<td>BIT #$0F</td>
</tr>
<tr>
<td>149</td>
<td>0000e080 e62f</td>
<td>BNE COL1</td>
</tr>
<tr>
<td>150</td>
<td>0000e083 b0c</td>
<td>COLRET BCS KEY2</td>
</tr>
<tr>
<td>151</td>
<td>0000e085 2503</td>
<td>IF VALID GET OUT</td>
</tr>
<tr>
<td>152</td>
<td>0000e087 5a</td>
<td>DECX ELSE TRY</td>
</tr>
<tr>
<td>153</td>
<td>0000e089 2646</td>
<td>BNE KEY1</td>
</tr>
<tr>
<td>154</td>
<td>0000e08c 81</td>
<td>NEXT ROW</td>
</tr>
<tr>
<td>155</td>
<td>0000e08d a60a</td>
<td>KEY2 RTS</td>
</tr>
<tr>
<td>156</td>
<td>0000e08f b72f</td>
<td>DBOUNC LDA #10</td>
</tr>
<tr>
<td>157</td>
<td>0000e091 a6ff</td>
<td>STA W6</td>
</tr>
<tr>
<td>158</td>
<td>0000e093 21fe</td>
<td>DLP LDA #$FF</td>
</tr>
<tr>
<td>159</td>
<td>0000e095 21fe</td>
<td>BRN *</td>
</tr>
<tr>
<td>160</td>
<td>0000e097 4a</td>
<td>BRN *</td>
</tr>
<tr>
<td>161</td>
<td>0000e099 26f9</td>
<td>DECA</td>
</tr>
<tr>
<td>162</td>
<td>0000e09b 342f</td>
<td>BNE DLOOP</td>
</tr>
<tr>
<td>163</td>
<td>0000e09c 26f3</td>
<td>DEC W6</td>
</tr>
<tr>
<td>164</td>
<td>0000e09e 81</td>
<td>BNE DLP</td>
</tr>
</tbody>
</table>

The keyboard routine returns the code of the pressed key in the accumulator.

The keyboard reads the code of the pressed key in the accumulator.

135
<table>
<thead>
<tr>
<th>Address</th>
<th>Code/Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000e09f</td>
<td>51 STABL FCB</td>
<td>$11 0 BRANCH OFFSET CALC.</td>
</tr>
<tr>
<td>0000e0a0</td>
<td>51 CTAB FCB</td>
<td>$51 P LOAD FROM EPROM ($4000)</td>
</tr>
<tr>
<td>0000e0a3</td>
<td>68 FCB</td>
<td>$68 S LOAD FROM EPROM ($8000)</td>
</tr>
<tr>
<td>0000e0a4</td>
<td>68 FCB</td>
<td>$68 S LOAD FROM EPROM ($8000)</td>
</tr>
<tr>
<td>0000e0a7</td>
<td>28 JMP</td>
<td>DUMP9</td>
</tr>
<tr>
<td>0000e0a8</td>
<td>36 FCB</td>
<td>cce3f8 PUNCH</td>
</tr>
<tr>
<td>0000e0ab</td>
<td>52 FCB</td>
<td>$52 L LOAD S-RECORD</td>
</tr>
<tr>
<td>0000e0ac</td>
<td>cce321 JMP</td>
<td>TLOAD</td>
</tr>
<tr>
<td>0000e0af</td>
<td>54 FCB</td>
<td>$54 V VERIFY (S-RECORD)</td>
</tr>
<tr>
<td>0000e0b0</td>
<td>cce31b JMP</td>
<td>VERIFY</td>
</tr>
<tr>
<td>0000e0b3</td>
<td>62 FCB</td>
<td>$62 G GO INTO EMULATOR MODE</td>
</tr>
<tr>
<td>0000e0b4</td>
<td>cce23d JMP</td>
<td>MODE3</td>
</tr>
<tr>
<td>0000e0b5</td>
<td>64 FCB</td>
<td>$64 M READ/CHANGE MEMORY</td>
</tr>
<tr>
<td>0000e0b8</td>
<td>cce4fd JMP</td>
<td>MEMEX</td>
</tr>
<tr>
<td>0000e0bb</td>
<td>38 FCB</td>
<td>$38 B ADDRESS OFFSET</td>
</tr>
<tr>
<td>0000e0bc</td>
<td>cce2c0 JMP</td>
<td>OFFSET</td>
</tr>
<tr>
<td>0000e0bf</td>
<td>48 LAST FCB</td>
<td>$48 A BRANCH OFFSET CALC.</td>
</tr>
<tr>
<td>0000e0c0</td>
<td>cce13e JMP</td>
<td>BRAOFF</td>
</tr>
<tr>
<td>0000e0c3</td>
<td>11 STABL FCB</td>
<td>$11 0 BRANCH OFFSET CALC.</td>
</tr>
<tr>
<td>0000e0c4</td>
<td>21 FCB</td>
<td>$21 1</td>
</tr>
<tr>
<td>0000e0c5</td>
<td>22 FCB</td>
<td>$22 2</td>
</tr>
<tr>
<td>0000e0c6</td>
<td>24 FCB</td>
<td>$24 3</td>
</tr>
<tr>
<td>0000e0c7</td>
<td>31 FCB</td>
<td>$31 4</td>
</tr>
<tr>
<td>0000e0c8</td>
<td>32 FCB</td>
<td>$32 5</td>
</tr>
<tr>
<td>0000e0c9</td>
<td>34 FCB</td>
<td>$34 6</td>
</tr>
<tr>
<td>0000e0ca</td>
<td>41 FCB</td>
<td>$41 7</td>
</tr>
<tr>
<td>0000e0cb</td>
<td>42 FCB</td>
<td>$42 8</td>
</tr>
<tr>
<td>0000e0cc</td>
<td>44 FCB</td>
<td>$44 9</td>
</tr>
<tr>
<td>0000e0cd</td>
<td>48 FCB</td>
<td>$48 A BRANCH OFFSET</td>
</tr>
<tr>
<td>0000e0ce</td>
<td>38 FCB</td>
<td>$38 B LOAD OFFSET</td>
</tr>
<tr>
<td>0000e0cf</td>
<td>28 FCB</td>
<td>$28 C OUTPUT S-RECORDS</td>
</tr>
<tr>
<td>0000e0d0</td>
<td>18 FCB</td>
<td>$18 D</td>
</tr>
<tr>
<td>0000e0d1</td>
<td>14 FCB</td>
<td>$14 E</td>
</tr>
<tr>
<td>0000e0d2</td>
<td>12 FCB</td>
<td>$12 F</td>
</tr>
<tr>
<td>0000e0d3</td>
<td>61 FCB</td>
<td>$61 10 Esc CANCEL COMMAND</td>
</tr>
<tr>
<td>0000e0d4</td>
<td>58 FCB</td>
<td>$58 11 E ENTER COMMAND</td>
</tr>
<tr>
<td>0000e0d5</td>
<td>68 FCB</td>
<td>$68 12 S LOAD FROM $8000</td>
</tr>
<tr>
<td>0000e0d6</td>
<td>64 FCB</td>
<td>$64 13 M MEMORY EXAMINE/CHANGE</td>
</tr>
<tr>
<td>0000e0d7</td>
<td>62 FCB</td>
<td>$62 14 G EMULATE</td>
</tr>
<tr>
<td>0000e0d8</td>
<td>54 FCB</td>
<td>$54 15 V VERIFY RAM</td>
</tr>
<tr>
<td>0000e0d9</td>
<td>52 FCB</td>
<td>$52 16 L LOAD RAM</td>
</tr>
<tr>
<td>0000e0da</td>
<td>51 FCB</td>
<td>$51 17 P LOAD FROM $4000</td>
</tr>
</tbody>
</table>
**Build a beginning and ending address**

**in TEMP, TEMP+1 & ADDRH, ADDRDL resp.**

```
PROCEDURE

BLDRNG  BCLR  4, STAT
        BCLR  2, STAT
        EMULATION ADDRESS

        JSR  CLRTAB
        PRINT

        LDA  #$F4
        'BA'

        STA  DTABL+4

        LDA  #$77

        STA  DTABL+5

        JSR  DISTAB

        JSR  BLDADR

        GET SOURCE ADDR.

        BCC  BLDRN1

        VALID?

        LDA  ADDRH

        YES

        STA  TEMP

        SAVE IT

        LDA  ADDRDL

        STA  TEMP+1

        JSR  CLRTAB

        PRINT 'EA'

        LDA  #$F1

        STA  DTABL+4

        LDA  #$77

        STA  DTABL+5

        JSR  DISTAB

        JSR  BLDADR

        GET DESTINATION ADDR

        BCC  BLDRN1

        VALID?

        LDA  ADDRH

        YES

        RTS

        BLDRN1

        BSET  4, STAT

        INVALID

        RTS

        JSR  STX

        W6

        CLR  COUNT

        LDA  $0F0

        LDX  W6

        STA  DTABL.X

        INC  W6

        INC  COUNT

        BLO  SLP

        JMP  OISTAB

        BLRTAB

        FCB  0.0, S00, S07, $77, S6E

        FCB  S06, SFF, S60, S06, S71, SBB

        VERF  FCB  0,$6,$F1,$60,$06,$71,$86

        OLOAD  FCB  0,$0,$D,$7,$77,$6E

        BLRTAB
```

**Display message.**
283
284
285
286
287
288

289 0000e13e ad9b
290 0000e140 08313e
291 0000e143 b62b
292 0000e145 a002
293 0000e147 b72b
294 0000e149 b630
295 0000e14b a200
296 0000e14d b730
297 0000e14f b62b
298 0000e151 b027
299 0000e153 b72b
300 0000e155 b630
301 0000e157 b226
302 0000e159 b730
303 0000e15b b62f
304 0000e15d a1lf
305 0000e15f 234e
306 0000e161 b630
307 0000e163 a1ff
308 0000e165 270b
309 0000e167 4d
310 0000e168 2674
311 0000e16a b62b
312 0000e16c a1ff
313 0000e16e 256e
314 0000e170 200a
315 316 0000e172 b62b
317 318 0000e174 a1ff
319 0000e176 2766
320 321 0000e178 a1ff
322 0000e17a 256e
323 0000e17c ad06
324 0000e17e cce000
325 326 0000e181 cce04e

em64k.as5

327 328
329 0000e184 cde235
330 0000e187 e606
331 0000e189 b720
332 0000e18b e6b5
333 0000e18d b721
334 0000e18f a6f1
335 0000e191 b722
336 0000e193 a666
337 0000e195 b723
338 0000e197 b62b
339 0000e199 cde5f2
340 0000e19c 97
341 0000e19d b627
342 0000e19f a001
343 0000e1a1 b72b
344 0000e1a3 b626
345 0000e1a5 e900
346 0000e1a7 b730
347 0000e1a9 9f
348 0000e1a a153

138
349 0000ELAC CCE562

350

351 0000E1AF B62B OFFST1 LDA ADDRL ADJUST FOR
352 0000E1B0 A001 SUB #1 BIT BRANCH
353 0000E1B3 B72B STA ADDRL
354 0000E1B5 B630 LDA ADDRH
355 0000E1B7 A200 SBC #0
356 0000E1B9 B730 STA ADDRH
357 0000E1BB A1FF CMP #$FF NEG OFFSET?
358 0000E1BD B70B BEQ OFFST3 YES
359 0000E1BF 4D TSTA CHECK FOR
360 0000E1C0 261C BNE OVRERR +/- 0 AND -1
361 0000E1C2 B62B LDA ADDRL
362 0000E1C4 A17F CMP #$7F
363 0000E1C6 2216 BHI OVRERR
364 0000E1C8 200A BRA OK2
365
366 0000E1CA B62B OFFST3 LDA ADDRL
367 0000E1CC A1FE CMP #$FE
368 0000E1CE 240E BHS OVRERR
369 0000E1D0 A180 CMP #$80
370 0000E1D2 250A BLO OVRERR
371
372 0000E1D4 3C27 OK2 INC TEMP+1
373 0000E1D6 2602 BNE OFFST
374 0000E1D8 3C26 INC TEMP
375 0000E1DA 8A2B OFFST BSR USE PRINT IF VALID
376 0000E1DC 200B BRA SCJMP
377
378 0000E1DE A6D7 OVRERR LDA #$D7 PRINT "OR"
379 0000E1E0 B724 STA DTABL+4
380 0000E1E2 A660 LDA #$60
381 0000E1E4 B725 STA DTABL+5
382 0000E1E6 CDE16 JSR PRTAOR
383 0000E1E9 CCE000 SCJMP JMP SCAN

em64k.a55

385

* Display table contents.
* * *

***************************************************************************

386

387

388

389

390

391 0000E1EC 1701 DISTAB BCLR 3.PORTB ENABLE (144115) LOW
392 0000E1EE AE05 LDX #5 LOAD DISPLAY
393 0000E1F0 E620 DISCHR LDA DTABL,X
394
395 0000E1F2 BFB8 NT1 STX WJ SAVE INDEX
396 0000E1F4 1000 BCLR 6.PORTA CLEAR DATA
397 0000E1F6 AE0B LDX #0
398 0000E1F8 4B DIS1 LSLA SET UP
399 0000E1F9 2402 BCC DIS2 BIT OF
400 0000E1FB 1C00 BSET 6.PORTA ACCUMULATOR
401 0000E1FD 1000 DIS2 BSET 7.PORTA CLOCK
402 0000E1FF 1000 BCLR 7.PORTA IT
403 0000E201 1000 BCLR 6.PORTA CLEAR DATA
404 0000E203 5A DECX COMPLETE?
405 0000E204 26F2 BNE DIS1 NO
406 0000E206 BE2B LDX W1 RESTORE INDEX
407 0000E208 5A DECX
408 0000E209 2AE5 BPL DISCHR
409 0000E20B 1601 BSET 3.PORTB ENABLE (144115) HIGH
410 0000E20D 81 RTS
411

139
412
413
414
415
416
417
418 0000e20e 1201
419 0000e210 0e3002
420 0000e213 1301
421 0000e215 1001
422 0000e217 0c3002
423 0000e21a 1101
424 0000e21c be30
425 0000e21e bf2a
426 0000e220 1c2a
427 0000e222 1f2a
428
429 0000e224 0a3108
430 0000e227 ae05
431 0000e229 bf29
432 0000e22b de05
433 0000e22d be05
434 0000e22f ae05
435 0000e231 ba05
436
437 0000e233 bc29

-------------------------------------------------------------
* S-record input RAM accessing.
*
-------------------------------------------------------------
RAMACC BSET 1,PORTB 5 5 XFER A14 & A15 TO PORTB
BRESET 7,ADDRH,A15H 5 10 A15 HIGH?
BCLR 1,PORTB 5 15 NO
A15H BSET 0,PORTB 5 20 YES
BRESET 6,ADDRH,A14H 5 25 A14 HIGH?
BCLR 0,PORTB 5 30 NO
A14H LDX ADDRH 3 33 YES
STX ADDEH 4 37
BSET 6,ADDEH 5 42 A14 HIGH
BCLR 7,ADDEH 4 47 A15 LOW

-------------------------------------------------------------
BRESET 5,STAT.L3 5 52 READING?
LDX #$C7 2 54 NO, WRITING (STA)
STX W2 4 58 STA IN
JSR W2 16 74 RAM SUBROUTINE
STA W4 4 78 SAVE FOR READBACK CHECK
STX W2 13 93 LDA IN RAM
JMP W2 14 107 121 (89 FOR READ) WITH JSR

-------------------------------------------------------------
Clear display table.
*
-------------------------------------------------------------
CLRTAB LDX #$5
CLRLOC CLR DTABL.X CLEAR SIX
DECK CLRLOC DISPLAY TABLE
RTS

-------------------------------------------------------------
Emulator mode.
*
-------------------------------------------------------------
MODE3 JSR CLRTAB
LDA #$F1 E
STA DTABL
LDA #$73 P
STA DTABL+1
LDA #$63 ?
STA DTABL+3
JSR DISTAB
JSR KEYSN WAIT UNTIL EPROM REMOVED
KSC JSR KEYSN EMULATION CONFIRMED?
BCC KSC
CMP #$62
BEQ CONF
JMP GETCMD

-------------------------------------------------------------
Conf LDA #$26 MODE 3, ENABLE (144115) HIGH
STA PORTB
LDA #$F1 E
STA DTABL
LDA #$06 U
STA DTABL+1
LDA #$D0 L

140
478 0000e269 b722 STA DTABL+2
479 0000e26a a677 LDA #$77 A
480 0000e26b b723 STA DTABL+3 T
481 0000e26c a6f0 LDA #$f0 E
482 0000e26d b724 STA DTABL+4
483 0000e26e a6f1 LDA #$f1
484 0000e26f b725 STA DTABL+5
485 0000e270 cdelec JSR DISTAB
486
487 0000e271 8e STP STOP

******************************************************************************
* Xfer EPROM contents to RAM from emulation socket ($4000).
* **************************************************************************

489
490
491
492
493
494
495
496 0000e27b cde235 DUMP1 JSR CLRTAB
497 0000e27c cdelec JSR DISTAB
498 0000e27d a658 LDA #$58 MODE 2. ENABLE (141)15 HIGH
499 0000e27e b701 STA PORTB REAL ADDRESS (NO OFFSET)
500 0000e27f 1431 BSET 2, STAT
501 0000e280 3f2b CLR ADDRL
502 0000e281 3f30 CLR ADDRH
503 0000e282 1201 LLPI BSET 1,PORTB 5 5 XFER A14 & A15 TO PORTB
504 0000e283 0e3002 BRSET 7,ADDRH,A015H 5 10 A15 HIGH ?
505 0000e284 1301 BCLR 1,PORTB 5 15 NO
506 0000e285 1001 BCLR 1,PORTB 5 15 NO
507 0000e286 0c3002 BRSET 6,ADDRH,A014H 5 25 A14 HIGH ?
508 0000e287 1101 BCLR 0,PORTB 5 30 NO
509 0000e288 be30 BCLR 0,PORTB 5 30 NO
510 0000e289 bf2a BCLR 0,PORTB 5 30 NO
511 0000e28a 1f2a BCLR 0,PORTB 5 30 NO
512 0000e28b 1c2a BCLR 0,PORTB 5 30 NO
513 0000e28c 1d01 BCLR 0,PORTB 5 30 NO
514 0000e28d 1e01 BSET 7,PORTB READ FROM EMULATOR SOCKET
515 0000e28e cde570 JSR LOAD LOAD BYTE
516 0000e28f 1c01 BSET 6,PORTB WRITE TO RAM
517 0000e290 1f01 BCLR 7,PORTB STORE BYTE
518 0000e291 cde562 JSR STORE
519 0000e292 3c2b INC ADDRL
520 0000e293 2602 INC ADDRH
521 0000e294 b630 SKPH LDA ADDRH
522 0000e295 6b30 SKPH LDA ADDRH
523 0000e296 2602 BNE LLP1 MSB ZERO ?
524 0000e297 26d2 BNE LLP1 YES. LSB ZERO ?
525 0000e298 b62b LDA ADDRH
526 0000e299 26ce BNE LLP1 IF SO, FINISHED
527

em64k.as5

141
OFFSET BSET 6.STAT NO ADDRESS INC/DEC
OFFSET BSET 2.STAT REAL ADDRESS
LDA #07 0
STA DTABL
LDA #71 F
STA DTABL+1
STA DTABL+2
CLR DTABL+3
CLR ADDEH
CLR ADDRH
LDA #0FF
STA ADDRl
JMP MEMEX3

----------

DUMP9 JSR CLRTAB
JSR DISTAB
LDA #58 MODE 2. ENABLE (14415) HIGH
STA PORTB
BSET 2.STAT REAL ADDRESS (NO OFFSET)

TLDP BSR T19
INC PORTB NEXT PAGE
LDA PORTB
AND #3
CMP #3 LAST PAGE ?
BLO TLP9
BSR T19 YES
JMP GETCMD

T19 CLR ADDRL
CLR ADDRH
LLP9 LDS ADDRH YES
STX ADDEH
STA ADDH BSET 7.ADH A15 HIGH
BCLR 6.ADH A14 LOW
JMP LOAD READ FROM AUXILIARY SOCKET
BCLR 7.ADH A15 LOW
BSET 6.ADH A14 HIGH
JSR STORE WRITE TO EMULATION RAM

142
<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>590</td>
<td>VERIFY BSET 5_STAT SERIAL VERIFY</td>
<td>*</td>
</tr>
<tr>
<td>591</td>
<td>LDX #6</td>
<td>*</td>
</tr>
<tr>
<td>592</td>
<td>BRA L4</td>
<td>*</td>
</tr>
<tr>
<td>593</td>
<td>TLOAD BCLR 5_STAT SERIAL LOAD</td>
<td>*</td>
</tr>
<tr>
<td>594</td>
<td>CLRX</td>
<td>*</td>
</tr>
<tr>
<td>595</td>
<td>L4 LDA #$81 RTS</td>
<td>*</td>
</tr>
<tr>
<td>596</td>
<td>STA W3</td>
<td>*</td>
</tr>
<tr>
<td>597</td>
<td>JSR DISP DISPLAY &quot;LOAD OR VERify&quot;</td>
<td>*</td>
</tr>
<tr>
<td>598</td>
<td>INPUT BSR INCHD 7 BIT ASCII INTO A</td>
<td>*</td>
</tr>
<tr>
<td>599</td>
<td>CMP #$5</td>
<td>NO. TRY AGAIN</td>
</tr>
<tr>
<td>600</td>
<td>BNE INPUT</td>
<td>YES, GET NEXT CHARACTER</td>
</tr>
<tr>
<td>601</td>
<td>CMP #$9</td>
<td>9</td>
</tr>
<tr>
<td>602</td>
<td>BNE NINE</td>
<td>YES, FINISH</td>
</tr>
<tr>
<td>603</td>
<td>CMP #$1</td>
<td>NO. 1</td>
</tr>
<tr>
<td>604</td>
<td>BNE INPUT</td>
<td>NO. TRY AGAIN</td>
</tr>
<tr>
<td>605</td>
<td>LENGHT CLR CHKSUM YES, CLEAR CHECKSUM</td>
<td>*</td>
</tr>
<tr>
<td>606</td>
<td>CLR TMP2 AND TEMP. STORE</td>
<td>*</td>
</tr>
<tr>
<td>607</td>
<td>JSR BYTEI AND GET BYTE COUNT</td>
<td>*</td>
</tr>
<tr>
<td>608</td>
<td>STA BCNT AND SAVE IT</td>
<td>*</td>
</tr>
<tr>
<td>609</td>
<td>ADDR JSR BYTEI ADDRESS HIGH</td>
<td>*</td>
</tr>
<tr>
<td>610</td>
<td>SUB OFF OFFSET</td>
<td>*</td>
</tr>
<tr>
<td>611</td>
<td>STA ADDRH</td>
<td>*</td>
</tr>
<tr>
<td>612</td>
<td>JSR BYTEI ADDRESS LOW</td>
<td>*</td>
</tr>
<tr>
<td>613</td>
<td>STA ADDRL</td>
<td>*</td>
</tr>
<tr>
<td>614</td>
<td>ADDR JSR BYTEI ADDRESS HIGH</td>
<td>*</td>
</tr>
<tr>
<td>615</td>
<td>SUB OFF OFFSET</td>
<td>*</td>
</tr>
<tr>
<td>616</td>
<td>STA ADDRH</td>
<td>*</td>
</tr>
<tr>
<td>617</td>
<td>JSR BYTEI ADDRESS LOW</td>
<td>*</td>
</tr>
<tr>
<td>618</td>
<td>STA ADDRL</td>
<td>*</td>
</tr>
<tr>
<td>619</td>
<td>DLOP JSR BYTEI 75 GET A BYTE</td>
<td>*</td>
</tr>
<tr>
<td>620</td>
<td>BEQ CHKX 3 78 LAST BYTE?</td>
<td>*</td>
</tr>
<tr>
<td>621</td>
<td>BRC LR 5_STAT.L5 5 83 NO VERIFYING?</td>
<td>*</td>
</tr>
<tr>
<td>622</td>
<td>STA W6 4 87 YES</td>
<td>*</td>
</tr>
<tr>
<td>623</td>
<td>JSR RAMACC 66 153 READ RAM</td>
<td>*</td>
</tr>
<tr>
<td>624</td>
<td>CMP W6 3 156 SAME?</td>
<td>*</td>
</tr>
<tr>
<td>625</td>
<td>BNE ERR7 3 159</td>
<td>*</td>
</tr>
<tr>
<td>626</td>
<td>BNE ERR7 3 159</td>
<td>*</td>
</tr>
<tr>
<td>627</td>
<td>BRA L6 3 162</td>
<td>*</td>
</tr>
<tr>
<td>628</td>
<td>JSR RAMACC 67 150 NO WRITE TO RAM</td>
<td>*</td>
</tr>
<tr>
<td>629</td>
<td>CMP W4 3 153 READBACK</td>
<td>*</td>
</tr>
<tr>
<td>630</td>
<td>BNE ERR2 3 156 OK?</td>
<td>*</td>
</tr>
<tr>
<td>631</td>
<td>L5 L6 INC ADDR 5 161 5 167 INCREMENT LS ADDRESS</td>
<td>*</td>
</tr>
<tr>
<td>632</td>
<td>BNE NDOVR 3 164 3 170 OVERFLOW?</td>
<td>*</td>
</tr>
<tr>
<td>633</td>
<td>BRA L5 3 172 3 178</td>
<td>*</td>
</tr>
</tbody>
</table>

em64k.as5

RS232 (9600) S-Record receiver (4MHz).
Checksum byte & error routine.

Checksum byte & error routine.

Input routine.

Cycles per bit at 9600 baud: 208
697
698
699
700
701
702
703 0000e3bd adcd
704 0000e3bf ad17
705 0000e3c1 4b
706 0000e3c2 4b
707 0000e3c3 4b
708 0000e3c4 4b
709 0000e3c5 b734
710 0000e3c7 b635
711 0000e3c9 bb32
712 0000e3cb b732
713 0000e3cd a8bd
714 0000e3cf ad07
715 0000e3d1 bb34
716 0000e3d3 b735
717 0000e3d5 3a36
718 0000e3d7 81
719
720 0000e3d8 a130
721 0000e3da 25d1
722 0000e3dc a139
723 0000e3de 2203
724 0000e3e0 a030
725 0000e3e2 81
726
727 0000e3e3 a141
728 0000e3e5 25ca
729 0000e3e7 a146
730 0000e3e9 22ca
731 0000e3eb a037
732 0000e3ed 81
733
734 0000e3ee a1ed
735 0000e3f0 2002
736 0000e3f2 a1e0
737 0000e3f4 5a
738 0000e3f5 26fd
739 0000e3f7 81

******************************************************************************

* Byte input sub-routines.
* *
* BYTE1 BSR INC10D 22 MS NIBBLE
BSR ASC11 35 57 WHAT WAS IT
LSLA 3 YES
LSLA 3 SHIFT
LSLA 3 IT
LSLA 3 69 UP
STA TMP1 4 71 AND SAVE IT
LDA TMP2 3 74 RESTORE BYTE
ADD CHKSUM 3 79 ACCUMULATE
STA CHKSUM 4 83 IN CHECKSUM BYTE
BSR INC10D 22 LS NIBBLE
BSR ASC11 35 57 WHAT WAS IT
ADD TMP1 3 60 ADD TO MS NIBBLE
STA TMP2 4 64 SAVE BYTE
DEC BCNT 5 69 DECREMENT BYTE COUNT
RTS 6 75

******************************************************************************

* ASCII CMP $30 2 BEFORE ZERO ?
BLO ERR3 3 5 YES, NOT LEGAL
CMP $39 2 7 AFTER NINE
BHI MT9 3 10 YES TRY A-F
SUB $30 3 13 0-9, CONVERT TO HEX
RTS 6 19

MT9 CMP $41 2 12 BEFORE A ?
BLO ERR4 3 15 YES, NOT LEGAL
CMP $46 2 17 AFTER F ?
BHI ERR5 3 20 YES, NOT LEGAL
SUB $37 3 23 A-F, CONVERT TO HEX
RTS 6 29

******************************************************************************

* DEL191 LDX $29 2
BRA DELAY 3 5
DEL110 LDX $16 2
DELAY DECK 3
BNE DELAY 3 6X
RTS 6 12+6X (INC BSR)

******************************************************************************

* PUNCH BSET 2,PORTBD BIT 2 OUTPUT
JSR BLDRNG BUILD RANGE
BRSET 4,STAT,NINE NEW ADDRESS ENTERED ?
LDX TEMP NO, SWAP ADDRESSES
STA TEMP
STX ADDRH
LDA ADDRL
LDX TEMP+1
STX ADDDR
STA TEMP+1
BCLR 7,STAT CLEAR END FLAG
BCLR 2,STAT EMULATION ADDRESS

******************************************************************************

* RS232 (9600 @ 4MHz) S-Record transmitter.
*
*
******************************************************************************

145
762 000e416 b72f STA W6  DIFFERENCE LSB
763 000e418 b626 LDA TEMP  END MSB
764 000e41a b230 SBC ADDRH  CURRENT MSB
765 000e41c 260d BNE LOTS  MSB ZERO?
766 000e41e b2bf LDA W6  YES, LOOK AT LSB
767 000e420 4c INCA  ADJUST
768 000e421 2708 BEQ LOTS  WAS $FF?
769 000e423 a120 CMP $20  MORE THAN 23?
770 000e425 2204 BHI LOTS  IF SO USE 23
771 000e427 1e31 BSET 7,STAT  NO. LAST SI RECORD
772 000e429 2002 BRA LTE20  LESS THAN OR EQUAL TO 20

774 000e42b a620 LOTS LDA $20  ADD BYTE COUNT & ADDRESS
775 000e42d a603 LTE20 ADD $503  ADD BYTE COUNT & ADDRESS
776 000e42f b736 STA BCNT  No. BYTES THIS SI RECORD
777 000e431 ad53 LDA $1'1'  1
778 000e433 cde48 JSR OUCH  JSR OUCH
779 000e436 a631 LDA #$1'  JSR OUCH
780 000e438 ad4a BSR OUCH  BSR OUCH
781 000e43a 3f32 CLR CHKSUM  CLR CHKSUM
782 000e43c b636 LDA BCNT  BYTE COUNT
783 000e43e ad32 BSR BYTEO  BSR BYTEO
784 000e440 b630 LDA ADDRH  ADDRESS HIGH
785 000e442 ad66 BSR BYTEO  BSR BYTEO
786 000e444 b62b LDA ADDRL  ADDRESS LOW
787 000e446 a60a BSR BYTEO  BSR BYTEO
788 000e448 cde570 LOOP2 JSR LOAD  GET BYTE
789 000e44b 3c2b INC ADDR  INCREMENT ADDRESS
790 000e44d 2602 BNE NOVR  OVERFLOW?
791 000e44f 3c30 INC ADDR  YES, INC. HIGH BYTE
792 000e451 ad5f NOVR BSR BYTEO  SEND BYTE
793 000e453 26f3 BNE LOOP2  LAST BYTE?
794 000e455 b632 BSR BYTEO  SEND IT
795 000e457 43 BSR CRLF  CRLF
796 000e459 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
797 000e45b a603 LDA #$03  BYTES
798 000e45d a61f LDA #$00  DUMMY(0)
799 000e461 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
800 a653 LDA $0'S'  S
801 000e465 593 a60a LDA #$0'9'  9
802 000e46a 3c30 BSR ADDR  YES, INC. HIGH BYTE
803 000e46f 7f30 BSR BYTEO  SEND IT
804 000e46e 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
805 a639 BSR BYTEO  SEND IT
806 000e470 603 BSR CRLF  CRLF
807 000e471 ad1d LDA #$03  3 BYTES
808 000e473 a647 LDA #$00  DUMMY(0)
809 000e474 3c30 BSR ADDR  YES, INC. HIGH BYTE
810 000e476 a600 BSR BYTEO  SEND IT
811 000e478 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
812 000e47a a653 LDA $0'S'  S
813 000e47e 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
814 000e486 5f30 BSR ADDR  YES, INC. HIGH BYTE
815 000e48c 7f30 BSR BYTEO  SEND IT
816 000e493 260d BNE NOVR  OVERFLOW?
817 000e495 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
818 000e49a a60a BSR BYTEO  SEND IT
819 000e49e 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
820 000e4a6 a60a BSR BYTEO  SEND IT
821 000e4a8 3c30 BSR ADDR  YES, INC. HIGH BYTE
822 000e4ab 603 BSR CRLF  CRLF
823 000e4a9 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
824 000e4ac a639 BSR BYTEO  SEND IT
825 000e4b1 ad1d LDA #$03  3 BYTES
826 000e4b3 a647 LDA #$00  DUMMY(0)
827 000e4b5 3c30 BSR ADDR  YES, INC. HIGH BYTE
828 000e4b8 a60a BSR BYTEO  SEND IT
829 000e4bc 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
830 000e4c5 a653 LDA $0'S'  S
831 000e4c7 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
832 000e4c8 5f30 BSR ADDR  YES, INC. HIGH BYTE
833 000e4cd 7f30 BSR BYTEO  SEND IT
834 000e4cf 260d BNE NOVR  OVERFLOW?
835 000e4d1 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
836 000e4e6 a60a BSR BYTEO  SEND IT
837 000e4e8 3c30 BSR ADDR  YES, INC. HIGH BYTE
838 000e4ec 603 BSR CRLF  CRLF
839 000e4f0 a639 BSR BYTEO  SEND IT
840 000e4f3 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
841 000e4f9 a653 LDA $0'S'  S
842 000e4fd 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?
843 000e4ff 5f30 BSR ADDR  YES, INC. HIGH BYTE
844 000e505 7f30 BSR BYTEO  SEND IT
845 000e50c 260d BNE NOVR  OVERFLOW?
846 000e50e 0f31b3 BRCLR 7,STAT,LOOP1  FINISHED?

******************************************************
*                                                      *
*      Checksum byte.                                   *
*                                                      *
******************************************************

796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827

******************************************************
*                                                      *
*      S9 record.                                       *
*                                                      *
******************************************************

146
Byte output sub-routine.

Output routine. 208 cycles per bit.

---

Output routine.

---

BYTEO STA TMP1
LSRA SHIFT
LSRA
LSRA GET MSB
LSRA
BSR ASCIO & CONVERT
BSR OUCH
LDA TMP1
ADD CHKSUM
STA CHKSUM
LDA TMP1
AND #$0F
LSB
BSR ASCIO CONVERT IT
BSR OUCH
DEC BCNT DECREMENT BYTE COUNT
RTS
Segment codes for the MC145000.

CTABL FCB $D7 0
       FCB $D6 1
       FCB $E3 2
       FCB $A7 3
       FCB $36 4
       FCB $85 5
       FCB $F5 6
       FCB $D7 7
       FCB $F7 8
       FCB $B7 9
       FCB $77 A
       FCB $F4 B
       FCB $D1 C
       FCB $E6 D
       FCB $F1 E
       FCB $71 F

ERROR JSR CLRTAB
       LDA #$F1
       STA DTABL+1
       LDA #$60
       STA DTABL+2
       STA DTABL+3
       JMP DSCN

INPUT ONE CHARACTER
* A REGISTER CONTAINS HEX VALUE
* X REGISTER CONTAINS HEX VALUE

CHRIN JSR KEYSCN GET KEY
       BCC CHRIN IF NOT VALID RETRY
       CMP STABL,X CONVERT
       BEQ CHRIN2 TO HEX
       INX
       BRA CHRIN1
       TXA IF CANCEL
       RTS

Memory examine/change.

MEMEX BCLR 2,STAT EMULATION ADDRESS
       JSR GETADR GET ADDRESS
       CMP #$10 ESCAPE ?
       BEQ MEMEX4

MEMEX3 BSR LOAD LOAD DATA
       JSR PRTOAT PRETEND IT
       JSR GETNYB GET NEW NIBBLE
       CMP #$10 ESCAPE ?
       BEQ MEMEX4
       CMP #$11 ENTER ?
959 0000e51b 271a BEQ AORI NC
960 0000e51c 271a BEQ AORDEC
961 962 0000e51e 0000e520 a113 CMP $13 MEMORY
963 0000e521 2208 BHI CMOMOL VAll D HEX
964 965 0000e51f 0000e521 cde5f2 MEMEX JSR PRTDAT PRINT IT
966 967 0000e522 272e BEQ AOROEC
968 969 0000e523 0000e526 a111 CMP IISOF IF VALID TRY AGAIN
970 971 0000e524 2208 BHI CMOMOL VAll D HEX
972 0000e525 25f8 BCS MEMEXI IF VALID TRY AGAIN
973 0000e526 0000e529 b629 LDA W2 RESTORE ACCA
974 975 0000e527 0000e52a ad34 BSR STORE YES STORE IT
976 977 0000e528 0000e52c 25d6 BCS MEMEX3 STORE VAllO
978 0000e529 0000e52e oc3125 AORINC BRSET 6.STAT.MEMEX4
979 0000e52a 3c2b INC AORL YES GOTO
980 0000e52b 2614 BNE MEMEX2 NO
981 0000e52c 0000e52f 20c8 BNE MEMEX3 REPEAT
982 0000e52d 0000e530 a113 MEMEX2 CMP I/SI3 M?
983 0000e52e 0000e531 2616 BNE MEMEX4 NO
984 0000e52f 0000e532 b629 LDA W2 RESTORE ACCA
985 986 0000e530 0000e533 3c2b INC AORL YES GOTO
987 988 0000e531 0000e534 2616 BNE MEMEX3 REPEAT
989 990 0000e532 0000e535 3c2b INC AORL YES GOTO
991 0000e533 0000e536 2616 BNE MEMEX3 REPEAT
em64k.as5
992 0000e534 0000e537 3c2b INC AORL YES GOTO
993 0000e535 0000e538 2616 BNE MEMEX3 REPEAT
994 0000e536 0000e539 b629 LDA W2 RESTORE ACCA
995 0000e537 0000e53a ad34 BSR STORE YES STORE IT
996 997 0000e538 0000e53b 25f8 BCS MEMEXI IF VALID TRY AGAIN
998 0000e539 0000e53c 0000e53d b629 LDA W2 RESTORE ACCA
999 1000 0000e53a 0000e53b 3c2b INC AORL YES GOTO
1001 0000e53b 0000e53c 2616 BNE MEMEX3 REPEAT
1002 0000e53c 0000e53d 3c2b INC AORL YES GOTO
1003 0000e53d 0000e53e 2616 BNE MEMEX3 REPEAT
1004 0000e53e 0000e53f 0000e540 b629 LDA W2 RESTORE ACCA
1005 0000e53f 0000e540 3c2b INC AORL YES GOTO
1006 0000e540 0000e541 2616 BNE MEMEX3 REPEAT
1007 0000e541 0000e542 3c2b INC AORL YES GOTO
1008 0000e542 0000e543 2616 BNE MEMEX3 REPEAT
1009 0000e543 0000e544 b629 LDA W2 RESTORE ACCA
1010 0000e544 0000e545 ad34 BSR STORE YES STORE IT
1011 0000e545 0000e546 25f8 BCS MEMEXI IF VALID TRY AGAIN
1012 0000e546 0000e547 a113 MEMEX2 CMP I/SI3 M?
1013 0000e547 0000e548 2616 BNE MEMEX4 NO
1014 0000e548 0000e549 3c2b INC AORL YES GOTO
1015 0000e549 0000e54a 2616 BNE MEMEX3 REPEAT
1016 0000e54a 0000e54b 3c2b INC AORL YES GOTO
1017 0000e54b 0000e54c 2616 BNE MEMEX3 REPEAT
1018 0000e54c 0000e54d b629 LDA W2 RESTORE ACCA
1019 0000e54d 0000e54e ad34 BSR STORE YES STORE IT
1020 0000e54e 0000e54f 25f8 BCS MEMEXI IF VALID TRY AGAIN
1021 0000e54f 0000e550 a113 MEMEX2 CMP I/SI3 M?
1022 0000e550 0000e551 2616 BNE MEMEX4 NO
1023 0000e551 0000e552 b629 LDA W2 RESTORE ACCA
1024 0000e552 0000e553 ad34 BSR STORE YES STORE IT
1025 0000e553 0000e554 25f8 BCS MEMEXI IF VALID TRY AGAIN
1026 0000e554 0000e555 a113 MEMEX2 CMP I/SI3 M?
1027 0000e555 0000e556 2616 BNE MEMEX4 NO
1028 0000e556 0000e557 3c2b INC AORL YES GOTO
1029 0000e557 0000e558 2616 BNE MEMEX3 REPEAT
1030 0000e558 0000e559 b629 LDA W2 RESTORE ACCA
1031 0000e559 0000e55a ad34 BSR STORE YES STORE IT
1032 0000e55a 0000e55b 25f8 BCS MEMEXI IF VALID TRY AGAIN
1033 0000e55b 0000e55c a113 MEMEX2 CMP I/SI3 M?
1034 0000e55c 0000e55d 2616 BNE MEMEX4 NO
1035 0000e55d 0000e55e b629 LDA W2 RESTORE ACCA
1036 0000e55e 0000e55f cce04e JMP GETCMO
1037 0000e55f 0000e560 0000e561 0000e562 aec7 STORE LDX #$C7 SET-UP
1038 0000e561 0000e562 0000e563 a00c BSR LDSTCM ROUTINE
1039 0000e562 0000e563 0000e564 b72d STA W4 TO DO
1040 0000e563 0000e564 0000e565 a006 BSR LOAD TWO BYTE
1041 0000e564 0000e565 0000e566 b12d CMP W4 STORE
1042 0000e565 0000e566 0000e567 99 BEQ STRTS SEC
1043 0000e566 0000e567 0000e568 81 STRTS RTS
1044 0000e567 0000e568 0000e569 aec6 LOAD LDX #$C6 SET-UP ROUTINE
1045 0000e568 0000e569 0000e56a bfc2 LDSTCM STX W2 TO DO
1046 0000e569 0000e56a 0000e56b a81 LDX #$81 TWO BYTE
1047 0000e56a 0000e56b 0000e56c b72e STX W3 LOAD
1048 0000e56b 0000e56c 0000e56d 043120 BRSET 2.STAT.NORM REAL ADDRESS?
1049 0000e56c 0000e56d 0000e56e 1201 RMCC BSET 1.PORTB 5 5 XFER A14 & A15 TO PORTB
1050 149
Build a byte.

Get one character into ACCA

Build address A.X dest in

Get address ADR/L/ADRH. C set if new.

Get character

Build address A,X dest... address

in ADDRL/ADDRH. C set if new.

Get high address

Put char away

Print new address

150
<table>
<thead>
<tr>
<th>Line 1147</th>
<th>1148</th>
<th>1149</th>
<th>1150</th>
<th>1151</th>
<th>1152</th>
<th>1153</th>
<th>1154</th>
<th>1155</th>
<th>1156</th>
<th>1157</th>
<th>1158</th>
<th>1159</th>
<th>1160</th>
<th>1161</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1148</td>
<td>1149</td>
<td>1150</td>
<td>1151</td>
<td>1152</td>
<td>1153</td>
<td>1154</td>
<td>1155</td>
<td>0000ffe4 e022</td>
<td>1156</td>
<td>0000fff6 e022</td>
<td>1157</td>
<td>0000ffff e022</td>
<td>1158</td>
</tr>
</tbody>
</table>

* * *

**MC68HC05EO Vectors.**

* * *

<table>
<thead>
<tr>
<th>ORG</th>
<th>$FFF4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FDB</th>
<th>START</th>
<th>SERIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDB</td>
<td>START</td>
<td>TIMER B</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>TIMER A</td>
</tr>
<tr>
<td>FDB</td>
<td>GETCMD</td>
<td>EXTERNAL INTERRUPT</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>SWI</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>RESET</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>RESET</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>RESET</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>RESET</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>RESET</td>
</tr>
<tr>
<td>FDB</td>
<td>START</td>
<td>RESET</td>
</tr>
</tbody>
</table>
Driving LCDs with M6805 Microprocessors

By Peter Topping
MCU Applications Group
Motorola Ltd, East Kilbride

INTRODUCTION

M6805 microprocessors include a wide range of parts with a large diversity of on-chip features. These include A/D and D/A convertors, serial interfaces, timers and display drivers. The display drive capability of the microprocessors range from none beyond I/O pins, through high current ports, to specialised display drivers for LCDs and vacuum fluorescent displays.

The MC68HC05M series have vacuum fluorescent drive capabilities up to 40V. The MC68HC05L series include LCD drivers with capabilities ranging from the MC68HC05L6 (3 or 4 backplanes and 24 frontplanes) through the MC68HC05L7/9 with 8 or 16 backplanes and 60/40 frontplanes. The L9's 40 frontplanes can be expanded to 205 with three MC68HC68L9 expanders.

Microprocessors without special LCD circuitry can be used to drive single backplane LCDs directly but require regular software intervention if the requirement that the display receives only AC drive is to be met. Alternatively display driver chips can be used to interface microprocessors with single and multiple backplane displays.

This application note gives hardware and software examples for these different arrangements. The same methods also apply to other families of microprocessors, eg M6801 and M68HC11. The examples are arranged in the order of the number of backplanes.

SINGLE-BACKPLANE DISPLAYS

Single-backplane displays are commonly used where the number of segments required is limited, usually using the 7-segment format. They have the advantages over multiplexed displays of superior contrast and viewing angle and a wider range of operating voltage and temperature. They can be driven directly by microprocessors with the number of segments limited simply by the number of available pins which are (or can be configured as) outputs. An output pin is required for the backplane together with one for each segment. The ports are loaded with the segment data corresponding to the required display, as with any other peripheral being directly driven by I/O lines. In this case, however, the microprocessor must complement the signals (backplane and frontplanes) at regular intervals, thus satisfying the requirement that the display receives an AC waveform with only a small DC component. It is possible for interrupts to alter the timing of these voltage reversals and the programmer must ensure that the resultant DC component does not exceed that above which the life of the display is reduced.

An alternative method of driving single-backplane displays from microprocessors is to use an LCD driver. Figure 1 shows a 6-digit 7-segment circuit using 3 MC144115P LCD drivers. These chips are driven serially and constitute a simple shift register giving the programmer full control over the display.
They can also be simply cascaded to drive a display of any required size. Clearly, the number of chips and interconnections increases directly as the number of segments. This limits the practical size of a display using this arrangement. The output pin to segment connections can be chosen to suit the application. The arrangement used here has been chosen to be compatible with the 4-backplane MC145000 driver used in a later example.

The MC144115 has a three-line serial interface consisting of clock, data, and chip enable. The clock and data lines can be shared with other peripherals, provided that each peripheral has a separate enable line. The enable line can, however, be derived from the clock if no other chips share the clock and data. This method of saving an I/O line is used in application note ANE416. The MC144115 software example (listing 1) has been modified from the routine used in ANE416.

Figure 1. Single-backplane LCD display with MC144115P display drivers
LISTING 1

**************
\* Example program for MC144115 driven single backplane display. \*
**************

PORTA EQU $00 PORT A ADDRESS
PORTB EQU $01 - R -

ORG $0050
R RMB 6 WORKING NUMBER

TMP1 RMB 1 POSITION OF LSB
TMP2 RMB 1 POSITION OF MSB
TMP3 RMB 1
TMP4 RMB 1

ORG $1000

**************
\* First part of the display subroutine gets the segment codes corresponding to the BCD data for display. \*
**************

DISP LDA #05
        STA  TMP3
        LDX  TMP1 LSB

D3 LDA 0.X
        STX  TMP4
        TAX
        LDA  STAB.X FIND 7 SEGMENT CODE
        LDX  TMP3
        STA  R.X PUT IN DISPLAY TABLE
        DEC  TMP3
        LDX  TMP4
        DECX
        CPX  TMP2 FINISHED ?
        BNE  D3

1 2
3 4
5 6
7 8
9 00000000
10 00000001
11
12
13
14 00000050
15
16 00000056
17 00000057
18 00000058
19 00000059
20
21
22
23
24
25
26
27
28
29
30
31
32 00001000 a605
33 00001002 b75b
34 00001004 be56
35 00001006 f6
36 00001007 bf59
37 00001009 97
38 0000100a d6103c
39 0000100d be58
40 0000100f e750
41 00001011 3e58
42 00001013 be59
43 00001015 5a
44 00001016 b357
45 00001018 26ec
The second part of the display routine sends the 48 bits required by the display driver.

```
OUTT  BCLR  3,PORTB ENABLE LOW
LDX  #5   SEND DISPLAY TABLE TO 144115
DISCHR LDA  R,X
DISPLY STX  TMP3  SAVE INDEX
BCLR  6,PORTA CLEAR DATA
LDX  #8
BCLR LOX  PORTA CLEAR DATA
DECX  COMPLETE?
BNE DIS1 NO
LDX TMP3  RESTORE INDEX
DECX
BPL DISCHR
BSET  3,PORTB ENABLE HIGH
RTS

LCD segment table.
```

```
STABL  FCB  $EB  0 SEGMENT
        FCB  $60  1
        FCB  $C7  2 CODES
        FCB  $E5  3
        FCB  $6C  4 FOR THE
        FCB  $AD  5
        FCB  $AF  6 MC145000/144115
        FCB  $EO  7
        FCB  $EF  8 LCD DRIVER
        FCB  $ED  9
```
THREE-BACKPLANE DISPLAYS

The MC68HC05L6 can drive 24 frontplanes and either 3 or 4 backplanes, the number of backplanes being selectable in software. The data to be displayed is arranged in the display RAM as shown in figure 2. Note that data sheet for the MC68HC05L6 (AD1254) shows this relationship wrongly.

It can be seen that each frontplane occupies a nibble in the 12-byte RAM. There is thus a simple relationship between RAM location and displayed digit on a 4-backplane 7-segment display (each 2 frontplane digit corresponds to one byte). With a 3-backplane display, however, each digit corresponds to 3 nibbles (1.5 bytes) so the software required to translate the required segments into display RAM data is more complex. Listing 2 shows a suggested method of doing this.

<table>
<thead>
<tr>
<th>LCD data latch 00</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>($00)</td>
<td>Bp1</td>
<td>Bp2</td>
<td>Bp3</td>
<td>Bp4</td>
<td>Bp1</td>
<td>Bp2</td>
<td>Bp3</td>
<td>Bp4</td>
</tr>
<tr>
<td></td>
<td>Fp 01</td>
<td>Fp 02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2  MC68HC05L6 back/frontplane pin to LCD data latch bit relationship

The table which translates the required character into segments contains 2 bytes per character, the middle nibble of the 3 required being repeated. This simplifies the code required to write to the display RAM by using one nibble if the character is intended for an even position in the display and the other for an odd position. Figure 3 shows the L6 – LCD segment arrangement used in this example.

When using a microprocessor without an LCD drive capability a separate display driver can be used to drive a multiplexed display. The example shown in figure 4 and listing 3 uses the ICM7231B 3-backplane driver. The ICM7231B requires each character to be addressed through pins A0, A1 and A2 and the appropriate data written to pins D0, D1, D2 and D3. This parallel control uses more I/O lines than the serial arrangement employed in MC145000/1 and MC144115 drivers.

The fact that data is accepted in HEX and encoded into segments by the driver simplifies the software but reduces the versatility of the display as only the driver’s 16 characters are available. The ICM7231B driver displays 0–9, E, H, L, P and blank while the ICM7231A displays 0–9, A, B, C, D, E and F.

As with any multiplexed LCD drive, the contrast is dependent on the supply voltage to the driver’s multiplexer circuitry. In the case of the ICM7231, contrast can be adjusted using the potentiometer on pin 2 (figure 4).

![8-digit Triplexed LCD](image)

Figure 3  MC68HC05L6 with a 3-backplane LCD
LISTING 2

• Example program using the MC68HC05L6
• to directly drive a 3-backplane display.

* * *

Examp Ie

program

using the

MC68HC05L6
to directly drive a 3-backplane display.

* * *

10 00000009 LADD EQU $0009 LCD ADDRESS REGISTER
11 0000000B LOAT EQU $000B LCD DATA REGISTER
12
13
14
15 00000050 ORG $0050 DISPLAY REGISTER
16 00000058 ORG $0100

16
17 0000005A ORG $0100
18 00000059 ORG $0100

* * *

19

20

21

22

LCD segment look-up table.

* * *

23

24

25

26

27

28

29 00000100 acce L6TAB FCB $AC,$CA 0
30 00000102 00c0 FCB $00,40 1
31 00000104 e48e FCB $E4,4E 2
32 00000106 e0ce FCB $00,4E 3
33 00000108 48c4 FCB $48,4C 4
34 0000010a e84e FCB $E8,4E 5
35 0000010c ec4e FCB $EC,4E 6
36 0000010e 80c8 FCB $80,3B 7
37 00000110 ecce FCB $EC,4E 8
38 00000112 e8ce FCB $E8,4E 9
39 00000114 40d4 FCB $40,0A - (A: CC CC)
40 00000116 ec0c FCB $EC,4E E (B: 6C 46)
41 00000118 4cc4 FCB $4C,3A H (C: AC OA)
42 0000011a 2c02 FCB $2C,32 L (D: 64 C6)
43 0000011c cc8c FCB $CC,3B P (E: EC 0E)
44 0000011e 0000 FCB $00,00 (F: CC OC)
Main loop for L6 directly driven LCD.

This subroutine assumes that Q contains HEX data for display. As each character requires 10 bytes (9 bits contained in 3 nibbles with 1 bit (backplane) in each nibble not used) each execution of the loop handles 2 characters.

******************************************************************************

START
CLR W2
LDA #80
STA LADD
4-backplane, fast charge enabled.

L6DP
LDA W2
Get HEX data.

AND #OF
Only lower nibble is relevant.

x 2 (two bytes per digit in table).

Initialise digit pointer.

First write to S09: Bus/LCD ratio = 256.

4-backplane, fast charge enabled.

Get HEX data.

Only lower nibble is relevant.

From this byte only the upper nibble is relevant, lower nibble is lost as upper nibble is shifted down.

Save nibble to be combined with first nibble of next digit.

Address of next digit in Q.

Get HEX data.

Only lower nibble is relevant.

x 2 (two bytes per digit in table).

From this byte only the lower nibble is relevant, upper nibble is lost as lower nibble is shifted up.

Combine nibble with last nibble of previous digit and send byte to LCD.

Of previous digit and send byte to LCD.

Next LCD data latch.

Get second byte from segment table.

and send it to LCD.

Next digit (three per loop).

Next digit (two per loop).

Finished?

If not, do next two digits.

Initialise digit pointer.

Get first byte from segment table.

Send it to LCD data latch.

Keep LCD on, move to next latch.

Get second byte of segment data.

From this byte only the upper nibble is relevant, lower nibble is lost as upper nibble is shifted up.

Combine nibble with last nibble of previous digit and send byte to LCD.

Send byte to LCD.

Next latch (three per loop).

Next digit (two per loop).

Now digit (three per loop).

Next digit (two per loop).

Finished?

If not, do next two digits.
Figure 4. 3-backplane LCD driven by an ICM7321
LISTING 3

******************************************************************************
* Example program using the ICM7231 driver and a 3-backplane display. *
*******************************************************************************

PORTD EQU $0001 PORT B DATA
PORTC EQU $0002 PORT C DATA

ORG $0050
Q RM8 B DISPLAY REGISTER

ORG $0100

******************************************************************************
* Display contents of Q. *
*******************************************************************************

Disp LDA PORTD CLEAR
AND $50 LS NIBBLE OF PORTD
STA PORTD IE DIGIT ADDRESS - 0

LDX #8

AGAIN LDA 0-1,X
STA PORTC
BCLR 3,PORTD LATCH
BSET 3,PORTD DIGIT
DECX
BEQ OUT DONE?
INC PORTD NO, GOTO NEXT DIGIT
BRA AGAIN

OUT CLR PORTC

RTS
FOUR-BACKPLANE DISPLAYS

As mentioned above, the MC68HC05L6 can drive a 4-backplane display with up to 24 frontplanes directly. The resultant 96 pixels could be used to drive 2 digits of an 5x8 dot matrix display, but with this number of segments most applications will use 7-segment or customised displays. A 7-segment display of up to 12 digits can be used. The software required is similar to, and simpler than, that shown for the L6 with a 3-backplane display. When it is required to drive a 4-backplane display using a microprocessor without an LCD drive capability, the MC145000 offers a versatile solution. Up to 6 digits (12 frontplanes) can be driven directly and more can be driven by the addition of one or more of the 18-pin MC145001 expanders, each adding 11 frontplanes. The example shown in figure 5 and listing 4 drives 6 digits, the software being very similar to that shown for the MC144115 single-backplane driver. This is the result of both chips having the same shift-register/latch architecture despite the actual output signals being quite different. The listing also shows a routine using an SCI rather than port lines.

A difference between the MC145000 and the MC144115 is that the MC145000 has no chip-enable input. It can share its data line with other peripherals but must have a dedicated clock so that the controller can supply data independently of other chips.

For applications requiring more than the 12 frontplanes made available by the MC145000, the MC145003/4 may be appropriate. They provide 32 frontplanes for use with a 4-backplane display, allowing up to 128 segments. The MC145003 and MC145004 are identical except for their serial protocol. The MC145004 has an IIC bus interface incorporating the usual acknowledge procedure associated with the IIC standard. The MC145003 is the same, except that there is no acknowledge and hence no associated clock cycle. The incoming data is automatically latched after 128 bits have been received. If, however, it is required that the data be latched at other times, an enable pin is available.

For applications where Vdd and Vlcd are connected together, the LCD contrast is adjusted by adjusting Vdd. If the data is coming from a chip with a higher supply voltage, the input pins may go higher than the supply voltage of the MC145003/4. This is allowed for the clock and data pins, but not recommended for the enable pin as its input protection circuitry may clamp the input voltage. It is therefore not advisable to use the enable pin if the MC145003/4 has a different Vdd from the chip supplying it with data. In applications not using this pin it can be left floating or tied high.

The example shown in figure 6 does not use the enable pin; the example software sends all 128 bits every time it is executed. The latching is thus performed automatically. The circuit shows 2 6-digit displays, each with 12 frontplanes. Any display or combination of displays with up to 32 frontplanes can be used with the software shown in listing 6, as all 128 bits are always sent. The 6 lines of code (45–50) are commented out for use with the MC145003; they are required for the MC145004.
Figure 5. MC145000 driving a 4-backplane LCD

Figure 6. MC145003/4 driving 4-backplane LCDs
LISTING 4

* Example program for MC145000 driven four-backplane display.

** PORTA EQU $00 PORT A DATA
BAUD EQU $00 SCI BAUD RATE REGISTER
SCL EQU $01 " CONTROL REG. No. 1
SCR2 EQU $0F " " " 2
SCSR EQU $10 " STATUS "
SDAT EQU $11 " DATA "

ORG $0050

R RMB 6 WORKING NUMBER
W2 RMB 1 POSITION OF LSB
W3 RMB 1
W4 RMB 1
W5 RMB 1
W6 RMB 1 POSITION OF MSB

ORG $1000

** First part of the display subroutine replaces BCD with segment codes.

**

DISP LDA $05
STA W4
LDX W2 LSB
D3 LDA 0.X
STX W5
TAX
D6 LDA STABL.X FIND 7 SEGMENT CODE
LDX W4
STA R.X PUT IN DISPLAY TABLE
DEC W4
DECX
CPX W6 FINISHED ?
BNE D3

164
The second part of the display routine sends the 48 bits required by the display driver. For comparison two routines are included, one using port A lines and a second using the SCI.

```assembly
OUTT LDX #5 SEND DISPLAY TABLE TO 144115/145000
DISCHR LDA R,X
DISPLY STX W3 SAVE INDEX
BCLR 6,PORTA CLEAR DATA
LDX #6
DIS1 LSRA SET UP
BCC DIS2 BIT OF
BSET 6,PORTA ACCUMULATOR
DIS2 BSET 7,PORTA CLOCK
BCLR 7,PORTA IT
BCLR 6,PORTA CLEAR DATA
DECX COMPLETE ?
BNE DIS1 NO
LDX W3 RESTORE INDEX
DECX
BPL DISCHR

SCI LCD driver interface.

```assembly
LDX #5 INITIALISE X
MORE LDA R,X FETCH DIGIT
BRCLR 7,SCSR,* WAIT UNTIL TDRE = 1
STA SDAT WRITE IT TO SCI TX REG.
DECX NEXT DIGIT
BPL MORE DONE ?
BRCLR 6,SCSR,* WAIT UNTIL TC=1
RTS

STABL FCB $EB 0 SEGMENT
FCB $60 1
FCB $C7 2 CODES
FCB $E5 3
FCB $6C 4 FOR THE
FCB $AD 5
FCB $AF 6 MC145000/MC144115
FCB $E0 7
FCB $EF 8 LCD DRIVER
FCB $ED 9
```
LISTING 5

******************************************************************************
*                      Example program using MC145003/4 LCD Drivers.          *
******************************************************************************

IICP EQU $02        PORTC
IICDD EQU $06        PORTC
SCL EQU $06          IIC - clock line
SDA EQU $07          IIC - data line
DIN EQU $40          INPUT DATA
OUT EQU $C0          OUTPUT DATA

ORG $0050

W1 RMB 1
DPNT RMB 1            IIC WRITE POINTER
ADDR RMB 1             IIC ADDRESS
IIC RMB 16             IIC BUFFER (128 BITS)

ORG $0800

START STA ADDR
LDA $17
STA W1
LDA #ADDR
STX DPNT
SLOOP LOX DPNT DATA BUFFER POINTER
LDA 0.X
GET A BYTE
LDA #8
B BITS TO SHIFT
SLOOP ROLA
BCC ZERO
BIT = 0 ?
BSET SDA,IICP
NO, BIT = 1
DZERO BSET SCL,IICP
CLOCK HIGH
BCLR SCL,IICP
CLOCK LOW
BCLR SDA,IICP
DATA LOW
DECX
SLOOP LAST BYTE
INC DPNT
NEXT BYTE
STA SIDDD
BACK TO AN OUTPUT
BNE SLOOP
LAST BYTE ?

I2CEND BSET SCL,IICP
STOP CONDITION
BSET SDA,IICP
DATA GOES HIGH WHILE CLOCK HIGH

166
8/16-BACKPLANE DISPLAYS

For dot-matrix displays, 8 or 16 backplanes are common. This is the result of the large number of pixels required. A compromise between pin-count and contrast is made to decide the number of backplanes. The minimum pin requirement for a display with N segments would require the number of backplanes to be the square root of N, but with typical requirements of many hundred or thousands of pixels this is not practical as the resultant contrast would not be acceptable. Typical compromises are 8 or 16 backplanes, as this gives acceptable contrast and fits conveniently with the 8x5 dot-matrix format commonly used for this type of display.

The MC68HC05L7 and L9 are designed to directly drive this type of display. The L7 has 16 backplanes and 60 frontplanes allowing it to drive up to 960 pixels or 24 8x5 dot matrix digits (12 with x 8 multiplexing). The L9 has only 40 frontplanes (16 8x5 digits) but is capable of being used with MC68HC68L9 LCD drive expanders. Each MC68HC68L9, up to 3 of which may be added, contributes 55 frontplanes. An L9 and 3 expanders has thus 205 frontplanes allowing then to drive up to 3280 pixels or 82 8x5 dot matrix digits.

The display RAM contains a 5-bit word for each row of dots in the 8x5 format; thus, 8 locations are used for each digit, allowing easy addressing. The RAM corresponding to the digits driven by expanders is contained in the expanders, but appears in the L9's memory map as the data and address buses from the L9 to external memory are also used by the MC68HC68L9s.

Application note ANHK10/D shows an application using the L9 and also describes a method of extending the display size beyond that normally available using this device.

APPLICATION NOTES

The following application notes give complete applications using the type(s) of display indicated.

ANE404 An extended MC146805E2 CBUG05 system using the MC68HC25.
MC145000-driven 4-backplane, 6-digit and ICM7231B-driven 3-backplane, 8-digit display.
ANE416 MC68HC05B4 Radio Synthesizer.
MC145000-driven 4-backplane, 6-digit and MC144115-driven 1-backplane, 6-digit display.
ANE425 Use of the MC68HC68T1 RTC with M6805 Microprocessors.
ICM7231B-driven 3-backplane, 8-digit display.
ANHK10 The summary of the MC68HC05L9 Micro.
MC68HC05L9/MC68HC68L9-driven dot matrix display.
The following list shows some LCD driver devices. They are most suitable for 7-segment, 16-segment and custom displays. The 7SD column shows how many 7-segment digits each device can drive. With the possible exceptions of the MC145000/1 and the MC145003/4, they are not generally suitable for dot-matrix displays which have 35-40 segments per digit.

<table>
<thead>
<tr>
<th>Device</th>
<th>Back</th>
<th>Front</th>
<th>7SD</th>
<th>Drive</th>
<th>Expan.</th>
<th>Pins</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC14543</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>parallel</td>
<td>parallel</td>
<td>16</td>
<td>BCD</td>
</tr>
<tr>
<td>MC14544</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>parallel</td>
<td>parallel</td>
<td>18</td>
<td>ripple blank</td>
</tr>
<tr>
<td>MC144115</td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>3-line</td>
<td>yes</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>MC144117</td>
<td>2</td>
<td>16</td>
<td>4</td>
<td>3-line</td>
<td>no</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>MC145000</td>
<td>4</td>
<td>12</td>
<td>6</td>
<td>2-line</td>
<td>MC145001</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>MC145001</td>
<td>(4)</td>
<td>11</td>
<td>5.5</td>
<td>2-line</td>
<td>n/a</td>
<td>18</td>
<td>expander</td>
</tr>
<tr>
<td>MC145003</td>
<td>4</td>
<td>32</td>
<td>16</td>
<td>2- or 3-line</td>
<td>parallel</td>
<td>52</td>
<td>2- or 3-line</td>
</tr>
<tr>
<td>MC145004</td>
<td>4</td>
<td>32</td>
<td>16</td>
<td>2-line</td>
<td>parallel</td>
<td>52</td>
<td>IIC</td>
</tr>
<tr>
<td>MC145453</td>
<td>1</td>
<td>33</td>
<td>4</td>
<td>2-line</td>
<td>parallel</td>
<td>40</td>
<td>also 44-pin</td>
</tr>
<tr>
<td>ICM7231</td>
<td>3</td>
<td>24</td>
<td>8</td>
<td>parallel</td>
<td>no</td>
<td>40</td>
<td>BCD</td>
</tr>
</tbody>
</table>
MCM2814 Gang-programmer using an MC68HC805B6

By Peter Topping
MCU Applications Group
Motorola Ltd, East Kilbride

INTRODUCTION

Non-volatile memories of the type MCM2814 are widely used in consumer equipment to store semi-permanent, user-definable information. One of the most common applications is TVs. In a TV the NVM is used to store the channel number or frequency associated with each program number and may also store other information about the program (for example, fine-tuning and transmission standard). The NVM will also contain the optimum settings for the sound and picture analogue values. In some sets other data may be stored, for example, user-defined names for some or all of the available channels. In a production environment the initial loading of this type of information can be done quickly by copying an existing NVM. This application note describes a programmer, shown in Figure 1, which can perform this function. In four seconds it can fully program 8 MCM2814s in parallel and verify them individually. The programmer is controlled by an MC68HC805B6 microprocessor.

The B6 is ideally suited to this application as it has a 256 byte NVEEPROM, the same size as the MCM2814. The contents of a master MCM2814 can thus be loaded into the B6 and will remain there until a change of data is required. Both the B6 and the MCM2814 have a byte in which some bits are dedicated to data protection so, in fact, only 255 bytes are used.

PRINCIPLE OF OPERATION

The programmer has been designed with an emphasis on ease of use and consequently has as few controls as possible. The only control which is used regularly is the “RUN” button S1. When it is pressed, the NVMs are powered-up and programmer operation starts. When it is released, power is disconnected from the NVMs but remains on within the programmer so that the LED indicators remain.

Three different operations are available.

The first requirement is to load data into the programmer. This procedure is selected by pressing and holding button S3, and started by pressing, and holding, the button S1. The contents of a master NVM in socket #0 will be loaded into the programmer; this takes about 12 seconds. The MC68HC805B6 in the programmer retains this data in its non-volatile EEPROM. During loading both LEDs at socket #0 are on and the rest are off. Once the B6’s EEPROM is written it is verified against the NVM in socket #0, a green LED indicating a pass and a red LED a fail. Sockets 1-7 should be empty during this procedure but if they contain an NVM they will also be checked. An attempt to perform this routine without an NVM in socket #0 will not destroy the data in the programmer as the software checks for an IIC acknowledge before overwriting the contents of the EEPROM within the MC68HC805B6. If S3 is pressed, the position of the slide switch S2 (program and verify or verify only) is not relevant. As S1 supplies power to the NVM sockets, it is important that it is held until the procedure is complete. Reliable results will not be obtained if S1 is released before the verification has finished.
The main programming sequence is selected by putting S2 in position P/V (program and verify). The NVMs should be placed in the sockets and button S1 pressed, and held, until programming and verification are complete (4 seconds). During programming, both red and green LEDs are on. The procedure clears the write-protect bytes and programs the NVMs in parallel using the data stored within the B6 in the programmer. It then verifies the NVMs individually, the results being shown on the LEDs (green for pass, red for fail and neither for no acknowledge).

Running only the verify routine can be selected by placing S2 in position “V”. The procedure is otherwise similar to programming. S1 should be held in until verification is finished (up to 1 second depending on results).

CIRCUIT

The NVMs are arranged in a matrix using their IIC chip-selects (pins 1 and 2). These pins configure the two least significant bits (1 and 2) of the recognised IIC address. Bit 0 determines if the device is receiving or transmitting. The software uses only the address 1010000x and so an NVM will only be addressed if both its chip-select pins are low. The matrix thus allows each NVM to be addressed individually by I/O lines using common software. Six port B I/O lines (0-6) are used for this purpose.

The NVMs (and the IIC pullups) are only powered up when S1 is closed, allowing them to be inserted and removed when they are powered down. When S1 is pressed, 5 volts is also applied to the RC circuit connected to the base of the BC107 NPN transistor. This supplies a base current to the transistor while the 10µF capacitor is charging. During this transient the transistor is on and the MC68HC05B6 is in reset. The program starts when the capacitor is charged and the transistor switches off. The MC34064 also connected to the reset pin ensures that the microprocessor is held in reset if the supply is below 4.75V. This will be the case during power-up and power-down. This is advisable in any system with EEPROM in order to prevent the possibility of it being corrupted during the rise or fall of Vdd.

The 33 ohm resistor in series with the supply to the NVMs limits the current if a device is plugged in the wrong way round. If this happens, the BC327 PNP transistor is turned on and the RED fault LED lit. The actual voltage on the VDD pins of the NVMs is monitored by bit 5 on port D. Thus the microprocessor can tell if there is a short or a device inserted the wrong way round. If this is the case the software aborts any attempt to program or verify NVMs.

Ports A and C are used for the pass and fail LEDs respectively, while the input-only port D has two pins used to read the positions of S2 and S3.

SOFTWARE

On reset, the program initialises the ports, IIC addresses and the RAM location (STAT) which is used for status flags. The state of bit 0 on port D is then checked. This will be low unless the button (S3), which selects the loading of data into the programmer, is pressed. If it is low, either program and verify or verify only is executed according to the level of bit 1 on port D which is connected to switch S2.

The programming routine (PROG) switches on all the LEDs, enables all the NVMs and clears their write-protect by writing $00 to address $FF. A loop which reads a byte from the B6 within the programmer and writes it to the NVMs is then performed 255 times. Writing to the NVMs is carried out without checking for an acknowledge, so the full procedure will take place even if all the NVM sockets are empty. This is the only routine which does not check for an acknowledge.

Only 255 bytes are transferred, as the last byte in the MCM2814 is reserved for the write-protect status. The B6 also has a reserved byte, but it is the first byte, so the correspondence of the addresses between the B6 and the NVMs is offset by one.

The verify routine (VERF) switches off all the LEDs and compares the data in the B6 with the data in each of the NVMs. It immediately fails a socket which does not return an IIC acknowledge. The RAM location COUNT contains the number of the socket being checked and is used to switch on the appropriate LED once each verify is complete.
The NVM sockets are arranged in a 2 x 4 matrix using their two IIC chip selects. The IIC address used is always $A1 ($A0 for writing) and thus an MCM2814 will only respond if both its chip-selects are low. During the programming routine all lines in the matrix are low, but during verification each chip is individually selected by a low on one row and one column. The required value to be sent to port B to achieve this is derived from the value in COUNT using the table TAB1.

If the LOAD routine is selected it performs a loop which does the opposite of the program loop. It transfers the contents of an MCM2814 into the EEPROM of the MC68HC805B6 within the programmer. It only looks at socket #0 and checks for an acknowledge before it proceeds to over-write the data in the B6.

The three routines WRT, RDALL and WRTAL were used during de-bug, are self-explanatory, and have been left in the program for interest and for future use if required. They are not used by the MC68HC805B6 but can be used if the program is run on a development system (eg an M68HC05EVM).

The IIC READ, SEND and WRITE routines are self contained and, by using ordinary I/O lines, could be used on any 68(HC)05 microprocessor. The READ sub-routine reads one byte using the IIC address in ADDR and the sub-address in SUBARD. The commented out lines (382-385) provide a simple method of also reading a second byte. Data is returned in IOBUF (and IOBUF+1 in the case of a 2-byte read). Clearly, the number of bytes can be increased by including more in-line code or by executing a loop.

The SEND sub-routine writes data on the IIC bus at IIC address ADDR starting at the address in RAM contained in the index register. The number of bytes sent is determined by the value stored the accumulator. If, as in this example, a sub-address has to be sent, then it should be in the RAM location pointed to by the index register with subsequent locations containing the data. SEND is suitable for most IIC peripherals. The WRITE subroutine constitutes a SEND followed by a delay and a READ. This is what is required to write a byte to an MCM2814, EEPROM programming is started by an IIC write and continues until the chip is addressed again by READ.

The IIC routines use simple BCLR and BSET instructions on the data registers so that an active high level is generated. This is not strictly what should be done on an IIC bus but is OK in the case where there are no other possible masters in the system. A passive high can be obtained by keeping the data bit of the I/O lines used for the IIC bus at a zero and using BCLR and BSET instructions on the corresponding data direction bits. If this is done care must be taken to ensure that these zeros are not lost by any other read-modify-write operations carried out on the data register. A BCLR or BSET on another bit on the same port will read the whole port, modify the required bit and then write the whole port. The data bit on an IIC line which was pulled high by the external pull-up would thus become a one. This would cause a malfunction the next time the corresponding DDR bit was set to a one, as the pin would output a high instead of a low. This can be avoided by using other types of instructions (eg STA) or by making sure the relevant data bit is a zero before enabling an output.
**MC68HC05B6 controlled MCM44182 programmer.**

* This software was developed by Motorola Ltd. for demonstration purposes.
* No liability can be accepted for its use in any specific application.
* Original software copyright Motorola - all rights reserved.
* P. Topping 14th May '91

```
PORTA EQU $00 PORT A ADDRESS
PORTB EQU $01
PORTC EQU $02
PORTD EQU $03
PORTAD EQU $04 PORT A DATA DIRECTION REG.
PORTBD EQU $05
PORTCDE EQU $06
EECTL EQU $07 EEPROM CONTROL REGISTER
E2PR EQU $0100 EEPROM
BUFF EQU $0000 DEBUG USE ONLY
```

**RAM allocation.**

```
ORG $0050
ADDR RMB 1 IIC ADDRESS
DPWT RMB 1 IIC DATA POINTER
SUBADR RMB 1 IIC SUB-ADDRESS
IOBUF RMB 2 IIC BUFFER
W1 RMB 1 W
W2 RMB 1 O
W3 RMB 1 R
W4 RMB 1 K
W5 RMB 1 I
W6 RMB 1 N
W7 RMB 1 G
COUNT RMB 1 LOOP COUNTER
STAT RMB 1 STATUS BYTE:-
0: not used
1: IIC R/W 1:READ, 0:WRITE
2: ACKNOWLEDGE OK
RMB 154 not used
STACK RMB 7 8 BYTES USED (0 INTERRUPTS
SP RMB 1 AND 4 NESTED SUBROUTINES)
```

**Main program.**

```
START LDA #$C0 CONTROL LINES LOW
STA PORTB
LDA #$FF
STA PORTA LEDS OFF
STA PORTC
STA PORTAD ALL OUT - GREEN LEDS
```

173
STA PORTCD          ALL OUT - RED LEADS
LDA #$3F            6,7 IN - IIC BUS
STA PORTBD          0-5 OUT - NVMS SELECTION
LDX $100            WAIT 230ms FOR MCM2814 SUPPLY
JSR PW              TO SETTLE
BRSET 5,PORTD,CONT  NVMS POWERED UP?
BRA *              NO, DO NOT PROCEED
CONT LDA #$5A0      YES, INITIALISE IIC ADDRESS
STA ADDR            EECTL
CLR EECTL           READ FROM EEPROM (B6)
CLR SUBADR          STAT
CLR STAT            

eeprom 0,PORTD,SKLD LOAD B6 EEPROM ?

JMP LOAD            YES

SKLD BRSET 1,PORTD,VERF NO, VERIFY ONLY?

*****************************************************************

B6 EEPROM ($0101-$1FF) -> 8 NVMS ($00 - $FE).

*****************************************************************

PROG CLR PORTA      GREEN LEADS ON
PROG CLR PORTC      RED LEADS ON
LDA #$5C            ENABLE ALL NVMS
STA PORTB
LDA #$5F            SUBADR = $FF
STA SUBADR
JSR READ            CLEAR POR WRITE PROTECT
CLRRA DATA = $00

BSR WR1 CLEAR WRITE PROTECT BYTE

CLR SUBADR

CLRX

OLoop LDA $2PR1+1,X GET BYTE AND

BSR WR1 SEND IT TO NVM

CPX $255            DONE?

BNE OLOOP

BRA VERF YES, VERIFY

WR1 STA IOBUF       NVMS
LDA "$2             No. BYTES TO SEND (INC. SUB-ADDRESS)
LDA #$SUBADR        IIC WRITE
LDX #$SUBADR        NEXT LOCATION
INC SUBADR
LDX SUBADR
RTS

NVM.A55

*******************************************************************************

* Verify.

* B6 EEPROM ($0101-$1FF) v. NVMs ($00 - $FE).

*******************************************************************************
0211 08e0 20 fe
0212 08e2 3f 52
0213 08e4 5f
0214 08e5 cd 09 92
0215 08e6 04 5d 13
0216 08eb b6 53
0217 08ed be 52
0218 08ef d1 01 01
0219 08f0 d1 01 01
0220 08f2 26 0a
0221 08f4 3c 52
0222 08f6 be 52
0223 08f8 a3 ff
0224 08fa 26 e9
0225 08fc 98
0226 08fd 81
0227 08fe 99
0228 08ff 81
0229 0900 ee ed eb e7 de dd TAB1 FCB SEE,SED,SEB,SE7,SDE,SDD,SDB,$D7

NVM.AS5
0232
0233
0234
0235
0236
0237
0238
0239 0908 a6 ee
0240 090a b7 01
0241 090c a6 fe
0242 090e b7 00
0243 0910 b7 02
0244
0245 0912 cd 09 92
0246 0915 04 5d 17
0247 0918 b6 53
0248 091a 12 07
0249 091c 14 07
0250 091e ad 12
0251 0920 cd 0a 2d
0252 0923 12 07
0253 0925 ad 0b
0254 0927 3c 52
0255 0929 be 52
0256 092b a3 ff
0257 092d 26 e3
0258
0259 092f cc 08 5e
0260
0261 0932 be 52
0262 0934 d7 01 01
0263 0937 10 07
0264 0939 cd 0a 33
0265 093c 3f 07
0266 093e b1

NVM.AS5
0268
0269
0270
0271
0272
0273
0274
0275
0276
0277
0278 093f 3f 52

VRF1 CLR SUBADR
VLOOP JSR READ GET A BYTE FROM NVM
BRSET 2,STAT,FAIL ACKNOWLEDGE OK ?
LDA IOBUF YES, CHECK DATA
LDX SUBADR
CMP E2PR+1,X COMPARE WITH B6 EEPROM BYTE
BNE FAIL SAME ?
INC SUBADR YES, CONTINUE
LDX SUBADR
CPX $255 FINISHED (255 BYTES) ?
INC SUBADR
BNE SUBADR YES, CONTINUE
LDA $EE SELECT POSITION 0
STA PORTB
LDA $FE POSITION 0 LEDS ON
STA PORTA
STA PORTC
JSR READ GET BYTE FROM NVM
BRSET 2,STAT,SKIP ACKNOWLEDGE OK ?
LDA IOBUF YES, GET DATA
BSET 1,EECTL SET EILAT
BSET 2,EECTL SET EIERA
BSR DOIT ERASE BYTE
JSR P10 WAIT 9.2 mS
BSET 1,EECTL SET EILAT TO WRITE BYTE
BSR DOIT
INC SUBADR
LDX SUBADR
CPX $255
BNE ILOOP
JSR VERF
LDOIT LDX SUBADR GET ADDRESS
STA E2PR+1,X LATCH DATA
BSET 0,EECTL START PROGRAMMING
JSR P20
CLR EECTL STOP
RST

************************************.*******************

Debug routines for use with EVM, not used in 805B6.
************************************.*******************

******************************************************************************
*NVM ($0, $200-$5FE) -> B6 EEPROM ($101-$1FF).*
******************************************************************************
0279 0941 5f
0280
0281 0942 d6 0f 00
0282 0945 12 07
0283 0947 14 07
0284 0949 ad e7
0285 094b cd 0a 33
0286 094e 12 07
0287 0950 ad e0
0288 0952 3c 52
0289 0954 be 52
0290 0956 a3 ff
0291 0958 26 e8
0292
0293 095a 20 fe
0294
0295
0296
0297
0298
0299
0300
0301 095c 3f 52
0302 095e a6 ee
0303 0960 b7 01
0304
0305 0962 cd 09 92
0306 0965 be 52
0307 0967 b6 53
0308 0969 d7 0f 00
0309 096c 3c 52
0310 096e 26 f2
0311
0312 0970 20 fe
0313
0314
0315
0316
0317
0318
0319
0320 0972 3f 52
0321 0974 a6 ee
0322 0976 bh 01
0323 0978 5f
0324
0325 0979 d6 0f 00
0326 097c cd 08 50
0327 097f 26 f8
0328
0329 0981 20 fe
0330
0331
0332
0333
0334
0335
0336
0337
0338 0001
0339 0005
0340
0341 0006
0342 0007
0343 007f
0344 00ff
0345 003f
0346
0347 0983 13 5d
0348 0985 b7 55

 CLRX
 LOOP2 LDA BUFF.X
 SET EILAT
 BSET 1,EECTL
 SET EIERA
 JSR P20
 BSET 1,EECTL
 SET EILAT TO WRITE BYTE
 INC SUBADR
 LDX SUBADR
 CFX #$255
 BNE LOOP2

 0295 0952 0947 12
 0296 0950 ad e7
 0297 0952 3c 52
 0298 0954 be 52
 0299 0956 a3 ff
 0300 0958 26 e8
 0301 095a 20 fe
 0302 095e a6 ee
 0303 0960 b7 01
 0304 0962 cd 09 92
 0305 0965 be 52
 0306 0967 b6 53
 0307 0969 d7 0f 00
 0308 096c 3c 52
 0309 096e 26 f2
 0310 0970 20 fe
 0311
 0312 0972 3f 52
 0313 0974 a6 ee
 0314 0976 b7 01
 0315 0978 5f
 0316 0979 d6 0f 00
 0317 097c cd 08 50
 0318 097f 26 f8
 0319
 0320 0981 20 fe

ANDROID

 page 7

***************************

IIC routines.

***************************

0338 0001
0339 0005
0340
0341 0006
0342 0007
0343 007f
0344 00ff
0345 003f
0346
0347 0983 13 5d
0348 0985 b7 55

IICF EQU $01
IICDD EQU $05
SCL EQU $06
SDA EQU $07
DIN EQU $7F
DOUT EQU $FF
OPEN EQU $3F
SEND BCLR 1,STAT

177
WRITE BCLR 1, STAT
WRITE TO NVM
STA W1
SAVE No. BYTES TO SEND
BSR IICBUS
JSR P10
9.2 ms NVM WRITE TIME
READ BSET 1, STAT
READ IIC DATA

IICBUS BCLR SCL, IICP
CLOCK LOW
BSET SDA, IICP
DATA HIGH
BSET SCL, IICP
CLOCK HIGH

LDA #DOUT
DRIVE
STA IICDD
BOTH

BCLR 0, ADDR
RW = 0 ALWAYS WRITE (SUB-ADDRESS)
SEND CHIP ADDRESS

IICD3 BCLR SDA, IICP
START CONDITION
BCLR SCL, IICP
DATA GOES LOW WHILE CLOCK HIGH
STX DPNT
CHIP ADDRESS OUT
BSR SHIFT

SUBADR
SET BIT 0 FOR READ
RDBUS1 LDA #SUBADR
WRITE SUB-ADDRESS
BSR SHIFT

RDBUSQ BSET 0, ADDR
SET BIT 0 FOR READ
BSET SDA, IICP
BOTH HIGH, NO
BSET SCL, IICP
STOP CONDITION
BCLR SDA, IICP
START CONDITION
BCLR SCL, IICP

BSR SHIFT
RE-SEND CHIP ADDRESS

RDBUS3 LDA #DIN
DATA IN FROM BUS
STA IICDD
BSR RDB
READ 8 BITS

RDBUS2 BSET SCL, IICP
CLOCK HIGH
BSR RACKF
DUMMY ACKNOWLEDGE
LDA IOBUF
MOVE
STA IOBUF+1
UP
BSR RDB
AND READ 8 MORE

RDB LDX #8
CLOCK HIGH
RDBUS2 BSET SCL, IICP
DATA LINE (RESULT IN CARRY)
BSET SDA, IICP, SCL, IICP
CLOCK LOW
RDBU SL
IOBUF
DECX
RDBUS2
BNE
RDSUB2
RTS
RACK

BSET SDA, IICP
LAST BYTE READ: SDA HIGH
RACK
BSR RACK

BRA IICEND

BSET SDA, IICP
DUMMY ACKNOWLEDGE
LDA IOBUF
MOVE
BSR SCL
CLOCK

BSET SDA, IICP
SDA OUT

BSET SDA, IICP
BSR SCL
CLOCK
<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0419</td>
<td>09e4 a6 7f</td>
<td>LDA #DIN SDA IN</td>
</tr>
<tr>
<td>0420</td>
<td>09e6 b7 05</td>
<td>STA IICDD</td>
</tr>
<tr>
<td>0421</td>
<td>09e8 81</td>
<td>RTS</td>
</tr>
<tr>
<td>0422</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0423</td>
<td>09f9 1f 01</td>
<td>RACKF BCLR SDA, IICP ACKN. WITH FOLLOWING BYTE</td>
</tr>
<tr>
<td>0424</td>
<td>09eb 20 ed</td>
<td>BRA RA2</td>
</tr>
<tr>
<td>NVM.AS5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0426</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0427</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0428</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0429</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0430</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0431</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0433</td>
<td>09ed be 51</td>
<td>WRBUS LDX DPNT DATA BUFFER POINTER</td>
</tr>
<tr>
<td>0434</td>
<td>09ef 6f</td>
<td>LDA 0,X DATA</td>
</tr>
<tr>
<td>0435</td>
<td>09f0 ad 0f</td>
<td>BSR SHIFT</td>
</tr>
<tr>
<td>0436</td>
<td>09f2 3c 51</td>
<td>INC DPNT</td>
</tr>
<tr>
<td>0437</td>
<td>09f4 3a 55</td>
<td>DEC W1 No. BYTES</td>
</tr>
<tr>
<td>0438</td>
<td>09f6 26 4f</td>
<td>BNE WRBUS</td>
</tr>
<tr>
<td>0439</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0440</td>
<td>09f8 1c 01</td>
<td>IICEND BSET SCL, IICP STOP CONDITION</td>
</tr>
<tr>
<td>0441</td>
<td>09fa 1e 01</td>
<td>BSET SDA, IICP DATA GOES HIGH WHILE CLOCK HIGH</td>
</tr>
<tr>
<td>0442</td>
<td>09fc a6 3f</td>
<td>LDA #OPEN TRI-STATE</td>
</tr>
<tr>
<td>0443</td>
<td>09fe b7 05</td>
<td>STA IICDD</td>
</tr>
<tr>
<td>0444</td>
<td>0a00 81</td>
<td>RTS</td>
</tr>
<tr>
<td>0445</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0446</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0447</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0448</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0449</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0451</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0452</td>
<td>0a01 ae 08</td>
<td>SHIFT LDX #8</td>
</tr>
<tr>
<td>0453</td>
<td>0a03 49</td>
<td>SHIFT1 ROLA</td>
</tr>
<tr>
<td>0454</td>
<td>0a04 24 04</td>
<td>BCC SHIFT2 SHIFT OUT 8 BITS 0 ?</td>
</tr>
<tr>
<td>0455</td>
<td>0a06 1e 01</td>
<td>BSET SDA, IICP NO, DATA = 1</td>
</tr>
<tr>
<td>0456</td>
<td>0a08 20 04</td>
<td>BRA SHIFT3</td>
</tr>
<tr>
<td>0457</td>
<td>0a0a 1f 01</td>
<td>SHIFT2 BCLR SDA, IICP DATA = 0</td>
</tr>
<tr>
<td>0458</td>
<td>0a0c 20 00</td>
<td>BRA SHIFT3 DELAY</td>
</tr>
<tr>
<td>0459</td>
<td>0a0e 1c 01</td>
<td>SHIFT3 BSET SCL, IICP CLOCK HIGH</td>
</tr>
<tr>
<td>0460</td>
<td>0a10 1d 01</td>
<td>BCLR SCL, IICP CLOCK LOW</td>
</tr>
<tr>
<td>0461</td>
<td>0a12 1f 01</td>
<td>BCLR SDA, IICP DATA LOW</td>
</tr>
<tr>
<td>0462</td>
<td>0a14 5a</td>
<td>DECX</td>
</tr>
<tr>
<td>0463</td>
<td>0a15 26 ec</td>
<td>BNE SHIFT1</td>
</tr>
<tr>
<td>0464</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0465</td>
<td>0a17 a6 7f</td>
<td>WACK LDA #DIN WRITE ACKNOWLEDGE</td>
</tr>
<tr>
<td>0466</td>
<td>0a19 b7 05</td>
<td>STA IICDD</td>
</tr>
<tr>
<td>0467</td>
<td>0a1b 15 5d</td>
<td>BCLR 2,STAT CLEAR FLAG</td>
</tr>
<tr>
<td>0468</td>
<td>0a1d 1c 01</td>
<td>BSET SCL, IICP CLOCK</td>
</tr>
<tr>
<td>0469</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0470</td>
<td>0a1f 0f 01 02</td>
<td>BRCLR SDA, IICP, ACOK ACKNOWLEDGE OK ?</td>
</tr>
<tr>
<td>0471</td>
<td>0a22 14 5d</td>
<td>BSET 2, STAT NO, SET FLAG</td>
</tr>
<tr>
<td>0472</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0473</td>
<td>0a24 1d 01</td>
<td>ACOK BCLR SCL, IICP</td>
</tr>
<tr>
<td>0474</td>
<td>0a26 1f 01</td>
<td>BCLR SDA, IICP</td>
</tr>
<tr>
<td>0475</td>
<td>0a28 a6 6f</td>
<td>LDA #DOUT</td>
</tr>
<tr>
<td>0476</td>
<td>0a2a b7 05</td>
<td>STA IICDD</td>
</tr>
<tr>
<td>0477</td>
<td>0a2c 81</td>
<td>RTS</td>
</tr>
<tr>
<td>NVM.AS5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0479</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0481</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0482</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0483</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0484</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0485</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0486</td>
<td>0a2d ae 04</td>
<td>P10 LDX #4 9.2 mS</td>
</tr>
</tbody>
</table>

page 9

Page 10

**Send sub-address and write data onto bus.**

**Shift out 8 bits and check acknowledge bit.**

**Delay (W2 x 2.3mS with a 2MHz bus).**
86 reset and interrupt vectors.

****************************************************

lff2 ORG $1FF2

0509
0510 lff2 08 00
0511 lff4 08 00
0512 lff6 08 00
0513 lff8 08 00
0514 lffa 08 00
0515 lffc 08 00
0516 lffe 08 00
0517
0518 END
"FLOF" Teletext using M6805 Microcontrollers

By Peter Topping
MCU Applications
Motorola Ltd, East Kilbride

1. INTRODUCTION

The "T" members of the MC68HC05 family of MCUs provide a convenient and cost effective method of adding on-screen-display (OSD) to TVs and VCRs. As well as the 64-character OSD capability, they include 8 Kbytes of ROM (adequate for Teletext, frequency-synthesis, stereo and OSD), 320 bytes of RAM, a 16-bit timer and 8 pulse-width-modulated D/A converters. The MC68HC05T7 also includes IIC hardware and, by using a 56-pin package, 4 ports of I/O independent of the OSD, serial and D/A outputs. It is thus suitable for large full-feature chassis. The MC68HC05T1 is in the middle of the price/performance range and includes most of the features of the MC68HC05T7 but in a 40-pin package. This is achieved by sharing I/O with the other pin functions (SPI, OSD, D/A). Even if all these features are used, there is sufficient I/O for most applications.

The MC68HC05T2 is a 16K upgrade of the MC68HC05T1 and the MC68HC05T3 a 24K version with increased RAM (512 bytes) and enhanced OSD (112 characters and 2 rows of OSD buffer). The low cost MC68HC05T4 has 5 Kbytes of ROM and 96 bytes of RAM making it suitable in simpler (eg mono, non-Teletext) applications. The T4 and T7 also include a 14-bit D/A converter to facilitate voltage synthesis tuning. There are EPROM (and OTP) versions of the T3 (including T1 and T2 emulation), T4 and T7.

This application note describes an example of Teletext control software written for the MC68HC05T7 which directly controls Teletext chips of the type 5243. Spanish FLOF Teletext (level 1.5) is handled using packet X/26. If no CCT teletext chip is present on the IIC bus (as indicated by the lack of an acknowledge), all Teletext functions are disabled in software. About 3Kbytes of ROM are used allowing the code to fit into the 7.9K bytes available in an MC68HC05T7 along with tuning, OSD and stereo functions.

The software in the included listing has been written for the MC68HC05T7 but could, with a little modification, be implemented on other M6805 microcontrollers. A microcontroller without IIC hardware can be used as long as additional software is included to facilitate the IIC bus using I/O pins. An example of IIC master I/O driven software can be found in application note AN446.

2. "FLOF" TELETEXT FEATURES

Full Level One Feature (FLOF) Teletext utilises "ghost" packets to provide features in addition to those available with the original CCT Teletext. The primary enhancement is the provision of a menu with a choice of four linked pages selectable by the user with a single press of one of four coloured buttons on the remote control. The menu itself is sent in the ghost page using packet 24 while the linked page numbers are contained in packet 27. In addition to linked pages, packets 26 and 30 are used. Packet 26 allows for the substitution of selected characters in the display by special characters specific to a particular country. This example application includes the Spanish implementation of packet 26. The broadcast service data packet (8/30) is used to get the initial (index) page for each channel and to display station identification information.
"Ghost" packets handled

X/24:

The FLOF menu information contained in this page extension packet is transferred by the microcomputer to row 24 of the display chapter. When links are disabled because there is no packet 27 (destination code 0) or when bit 4 of byte 43 is 0, row 24 is blank.

X/26:

Optional handling of modes 1xxxx, 01111 and 00010 in accordance with the Spanish Teletext specification. All the additional characters which are available in the 5243 CCT chip are handled. The feature can be disabled with a hardware link on an I/O pin (see figure 1) so that the software can be used at level 1.0 in non-Spanish countries also using packet 26.

X/27:

This packet contains the linked page numbers for the red, green yellow, blue and index (black) keys. Bit 4 on the link control byte (byte 43) is used to determine if these links are enabled (1) or disabled (0). When enabled, the Spanish specification requires that bits 1, 2 and 3 be used to enable the green, yellow and blue links respectively. This use of these bits is not defined in the World Teletext Specification. For this reason their use is selectable by a hardware link (see figure 1). If these bits are not used, all links (if enabled by bit 4) will be taken from packet 27 but will be automatically disabled if the broadcast links are default (FF3F7F) or invalid.

8/30:

The broadcast service packet is used to supply the index page number on exit from standby and (if teletext is not stopped) after a channel change. Bytes 10-30 of this packet are displayed for 5 seconds on exit from standby and (if teletext is not stopped) after a channel change.

3. IMPLEMENTATION

The software listing is in two parts. The first part contains the "idle" loop and IIC routines from the main TV control part of the MC68HC05T7 application. The idle loop controls the timing of everything performed by the microprocessor, scans the local keyboard, checks whether or not an IR command has been received, etc. It also monitors the relevant flags in the Teletext chip and performs the tasks (eg fetching linked pages) which have to be performed independently of requests for the user.

The second and main listing is the Teletext module itself. It contains all the subroutines required to carry out automatic and user requested Teletext activity. Both modules use the same RAM allocation file (RAMT8.S05) which is included in the listing of the Teletext module. This listing also includes a symbol cross-reference table.

Figure 1 shows a simplified circuit diagram of the application. Most of the MC68HC05T7's I/O is used for purposes other than Teletext and is not shown in detail. Communication with the 5243 Teletext chip is via an IIC bus in which the T7 is always the master. The function of the three I/O pins used for Teletext is described under "Ghost packets handled" and "Inputs and Outputs".

A version of this Teletext software has been implemented on an MC68HC05C4 for use in a TV where the other control functions were handled by a separate microcontroller. The signal from the IR pre-amp was fed into the C4 which used Teletext commands to control a 5243 via a software IIC bus. Non-Teletext commands were regenerated by the C4 and sent to the other microcontroller. This arrangement allows Teletext to be added to a chassis which was originally designed without considering Teletext.
Figure 1. MC68HC(7)05T7 – Teletext application circuit

4. IDLE LOOP

In the example application the idle loop code is in the main TV control software module rather than in the teletext module. Listing 1 shows the relevant parts of this module. The loop time is 12.8mS and it is at this rate that the timing counters used by Teletext (CNT1 and CNT4) are incremented. The standby condition is checked first; if the TV set is in standby then there is no IIC activity and hence no reading from, or writing to, the 5243. If the TV has just exited from standby, as indicated by the flag 3,STAT2, then Teletext is initialised using the sub-routine RESTRT. This sub-routine writes to the 5243’s control and mode registers (R5, R6 and R7) and checks that the IIC acknowledge is present. If there was no acknowledge, as indicated by flag 6,STAT7, then no further Teletext activity is attempted.

If an acknowledge is present, Teletext polling goes ahead, although it is suspended if there is a mute or time display. A mute indicates that the channel has just been changed, or no channel is tuned. During time display, all other Teletext activity is suspended. Re-initialisation using sub-routine START2 is performed if flag 7,STAT5 is set by a change of the tuned frequency.
Counter CNT4 is used to delay the transfer of packets 24 (page extension – FLOF menu), 27 (links), 26 (enhanced display characters) and the control bits from row 25 (display page) after the initial arrival of a page. When row 24 is read the 5243 FOUND flag is set to indicate that the arrival has been acted upon. If UPDATE is on then an update indicator appears if the update control bit (C9) is set or if the sub-page has changed or if it is the first arrival of the page. The update display is performed by the sub-routine ARRVD which clears the transient flags and enables the required display, i.e. page no. in normal mode and the whole of row 0 in sub-page mode. Any boxed information (eg sub-titles or newsflash) in the current page is also displayed. The last Teletext function performed by the idle loop is the checking of the FOUND flag in the 5243. This is accessed via the IIC bus; it is on the last (not displayed) row of the display page along with the current page and sup-page numbers and the control bits.

If there is a current Teletext transient (time, row 0 box or packet 8/30), the transient control branch from the idle loop is executed. This routine checks to see if it is time to end the transient. If it is, the subroutine OSDLE is executed. It resets transients for both the OSD generated by the MC68HC05T7 and Teletext. The sub-routine RSTMD2 performs this function for Teletext. It is called from within the sub-routine OSDLE (not listed).

5. REMOTE CONTROL FUNCTIONS

**TV/TXT**

Toggle between TV & Teletext mode.

**0-9**

Number keys for entry of page and sub-page numbers

**Red, Green, Yellow, Blue**

Linked page access keys. The decoder stores four pages of text. These are the display page and the three pages corresponding to the red, green and yellow links. The blue linked page is not acquired in advance. In the absence of FLOF data or if the links are disabled by the control bit in packet 27, the red key is page+1 and the green key page-1. Under these circumstances the requested page and the next three pages are acquired.

**PC+/−**

These keys always select page+1/page-1 regardless of the availability of FLOF information. As with the red, green and yellow keys, the page is displayed immediately if it is already in RAM.

**INDEX**

This key operates as an additional link with the difference that if the link is invalid the initial page from packet 8/30 is selected.

**SUB-PAGE/TIME**

Text mode: Enter sub-page mode, (max. 3979). TV mode: Display time in top-right-hand corner for 5 seconds. Pressing this key during a station identification display (packet 8/30 bytes 10-30) can be used to extend this display beyond the five seconds it appears for, after a channel change.

**STOP**

Halt acquisition, “STOP” is displayed instead of page number. Press again to restart. If acquisition has been stopped by partially entering a new page number then this key can be used to return to the original page.
MIX/NO-MIX

Toggle between Teletext and mixed display. Use of this key causes the display of the top status row for 5 seconds if it is not being displayed because the current page is a newsflash or a sub-title. 5243 contrast reduction is enabled in mixed mode.

FULL/TOP/BOT

Selects one of the three display formats, normal, top half enlarged, bottom half enlarged.

REVEAL

Reveal hidden text, toggle action.

UPDATE

Return to picture until a new version of the requested page arrives. When it arrives, its page no. is displayed in the top-right-hand corner, the key operates in both TV and Teletext mode, set is put into TV mode. Any boxed information (alarm clock, newsflash or sub-title) will be displayed. In sub-page mode the complete header is displayed so that both page & sub-page numbers can be seen. Cancel update by entering Teletext mode and then going back to TV mode by pressing the TV/Text key twice.

6. TELETEXT SUBROUTINES

6a. Subroutines: TVTX, UPDATE, DIGIT0 and GETIT

The Teletext module (listing 2) comprises various sub-routines which are used both by the idle loop and to perform any Teletext actions initiated by commands from the IR remote control. They are described in the order in which they appear in the listing.

TVTX is executed when the TV/TEXT button is pressed. Its function is to toggle between TV mode and Teletext mode. The flag 0,STAT indicates the current mode. This flag routes the microprocessor to execute either TXTOFF or TXTON according to the current mode. TXTON checks that Teletext hardware is present and does nothing if there has been no IIC acknowledge. If, however, a 5243 is present in the TV, it clears all transients (OSDLE) and sets up the Teletext mode. It initialises the control registers (R5 and R6) to display text and background both in and out of boxes. For newsflashes the set-up is text and background within boxes and picture outside. TXTOFF also resets transients but forces TV mode and sync. Polling and updating continue as a background activity.

When the UPDATE key is pressed the update flag 4,STAT2 is set and TXTOFF executed so the TV is forced to TV mode. If there is a current transient hold (eg time), the hold is cleared before TXTOFF is executed.

The number entry sub-routine DIGIT0 branches to DIGITS in sub-page mode but otherwise accepts any number key as a page number input. Three digits are required, the pointer PDP holding the current position (0, 1 or 2 for hundreds, tens or units). During entry the flag 2,STAT is set to stop Teletext activity. The numbers have to be written to the top-left-hand corner of the display page as well as saved in RAM. Once all three digits have been entered the page is requested and page acquisition restarted.

The code at label GETIT makes this request after first checking whether or not the selected page has already been requested (it could be the current display page or an already requested linked page). If it has, then a switch is made to the chapter associated with the appropriate acquisition circuit and no new request is generated. If not, the new request is made and the FOUND flag set.
6b. Subroutines: Colours, INDEX, NPAGE and PPAGE

The four colour keys (Red, Green, Yellow and Blue) are primarily intended for selecting Teletext linked pages. When pressed the chapter which corresponds to the appropriate acquisition circuit is selected for display. If links are disabled (by the link control bit or because there is no packet 27), then the RED and GREEN keys select current page +1 and -1 respectively. This choice is taken according to the state of flag 3, STAT3 which reflects the condition of the link control bit in packet 27. The code executed by RED, if links are not in use, is the same as that executed by the "+" function (NPAGE) which always selects the next page. Similarly the alternative GREEN function (PPAGE) is the same as for the "-" key. The YELLOW and BLUE keys do nothing under these circumstances. In Spanish Teletext the GREEN, YELLOW and BLUE links can be individually inhibited, but the RED link is only inhibited if all links are off.

The chapter associated with the selected page is displayed immediately if it has already been requested. This will normally be the case if a linked page (red, green or yellow) has been selected. The code at label LPT is executed if the page has already been requested. If not, a jump to CLRPD is performed. CLRPD is a label within DIGIT0; the code at CLRPD requests a new page just as if the page number had been entered manually. If the required acquisition circuit is the one already current, then the "unstop" code is executed. This causes the green page-being-looked-for header to roll as though the page number had just been entered. This means that something can be seen to happen in the case where the linked page differs only from the current page in its sub-page number. Linked sub-pages are not fully supported in this implementation as they are rarely used by broadcasters and would significantly increase the size of the software. When the chapter is changed the Teletext PBLF (page being looked for) flag is checked. If it is low the FOUND flag is cleared. This forces the fetching of the links associated with the new display page. If the page is not already in, this will automatically happen when it arrives so the FOUND flag does not need to be cleared.

The BLUE (or cyan) key is different in that its page will not normally be immediately available (the four pages: display, red, green and yellow occupy the four acquisition circuits and RAM chapters).

The INDEX (or black link) function is similar to BLUE except that if its link is not valid it defaults to the initial (index) page number supplied by packet 8/30 (see sub-routine GIP).

6c. Subroutines: LINK, GLP1, GLP2, SRCH, CHCK1 and NOTOKx

The sub-routine LINK allocates the three linked pages (RED, YELLOW and GREEN) to the three free acquisition circuits (not in use by the display page). To do this it checks the page numbers in turn to see if they have already been requested. If so they are left in their current acquisition circuit. If they have not already been requested the page number is put into a LIFO. Only 0-9 are regarded as acceptable digits for page numbers; this is consistent with the Spanish specification although the additional HEX numbers (A-F) may be used experimentally or by Teletext page generators. Within this first loop the sub-routine GLP1 is used to get the linked page numbers from packet 27, perform a decode of the Hamming encoded data and calculate the new magazine number (page hundreds) if different from that of the display page. GLP1 uses sub-routine SRCH to check if the page has already been requested. If there are no links, or if links are disabled, then displayed page +1, +2 and +3 are requested.

The second loop in LINK allocates new page numbers to the remaining unused acquisition circuits. It uses GLP2 to clear the relevant chapters in the Teletext memory and make the new requests. Subroutine CHCK1 is used to check whether or not an acquisition circuit is in use before it is loaded with a new page number from the LIFO.

This method of organising new page requests prevents unnecessary requests being made for pages already requested. This is particularly important when links are disabled and pages are being requested using the "+" or "-" functions. Under these circumstances when the page number is incremented (or decremented) only one new page has to be requested (new display page+3), while page, page+1 and page+2 do not need to change and can be left in their current acquisition circuits.
NOTOK3 and NOTOK2 handle the RED and GREEN functions when links are disabled. They are disabled if the link control bit (packet 27 bit3, byte 43) is zero or if there is no packet 27. These subroutines respectively increment and decrement the current page number (units and tens). The current magazine number (page hundreds) is not affected.

6d. Subroutines: ROW24, W2B, R2B, GCYI, CLINK and DECODE

ROW24 is used to transfer ghost row 20 (packet 24) into the display chapter. This has to be done via the IIC bus. The loop reads two bytes via the IIC (sub-routine R2B) bus from the ghost page and writes it to the display page (sub-routine W2B). The FOUND flag is then set to indicate that the arrival of the page has been recognised and acted upon. This sub-routine is only called by the idle loop and is used along with the other sub-routines which get information from the ghost page (CLINK, LINK and GET25).

R2B and W2B use IIC routines READ and SEND which are outwith the Teletext module. These subroutines will differ according to the microprocessor in use. An MC68HC05C8 implementation would need to use I/O lines (see reference for suitable software) while the MC68HC05T7 can use its IIC hardware. The routines used in this example are included in the listing extract from the TV control software module (listing 1).

The sub-routine GCYI is used by LINK to store the data associated with the BLUE an INDEX links. As explained above, these pages will not be acquired in advance, the page number only being sent to an acquisition circuit if requested by an IR command.

CLINK fetches the link control byte from packet 27 if the destination code is OK and, after decoding the Hamming encoded data, transfers the bits to STAT3.

The Hamming decode sub-routine DECODE corrects for single bit errors. This is done with in-line code using the table HAM (at the end of listing 2) as this uses less ROM than an algorithmic method.

6e. Subroutines: MIX, TRANx, TXTx, HOLD, and NOHOLD

The mixed display capability of the Teletext chip (5243) is toggled using an IR key which calls the sub-routine MIX. When mixed mode is entered, interlaced broadcast sync. (312/313) is selected because the non-interlaced sync. used for teletext is not suitable if a TV picture is present on the screen. This is set up via the 5243 mode register R1. The control registers R5 and R6 are updated to provide the mixed display.

When returning to a non-mixed display, the code at NOMIX is used to re-configure the control registers and to set up a Teletext only 312/312 non-interlaced sync. This sync. reduces adjacent line flicker in a pure Teletext display.

The subroutine TRAN2 sets up a transient which retains a black background on the top row so that the page number, time etc. can be seen clearly. This type of transient is also started if the page number or sub-page number is being entered in mixed mode. Sub-routines TRAN1, TRAN2 and TRAN3 are used to initialise the various transient displays. These displays are cancelled as discussed above by actions taken within the idle loop controlled by the free-running timer within the MC68HC05T7.

The TXTx sub-routines are used in conjunction with the IIC SEND routine to write to various sub-sets of the registers within the 5243.

If the Teletext STOP function is requested by an IR command the routine HOLD is executed. This is a toggled function when requested in this way. HOLD displays the word "STOP" in place of the page number and stops the display acquisition circuit by clearing the 5243 HOLD flag accessed via its page request register R3.

NOHOLD is executed to restart the display acquisition circuit. It returns the page number to the top-left-hand corner. If a new page number has been partially entered, a press of STOP (executing an UNHOLD) will allow a return to the most recent page request. This takes only a single press as the start of the entry of a new page number cause a HOLD. The completion of a page number entry (3 digits) causes a NOHOLD.
6f. Subroutines:  REVEAL, EXPTB and TIME

The REVEAL function causes any hidden display information to appear. It is controlled by a bit in the display mode register (R7). The software example leaves any revealed information permanently displayed. If, however, it is required that such information disappear when the page is updated (this may be better for a quiz page), then the two commented out lines (80 and 81) in the idle loop should be enabled.

The display expand facility is controlled by another two bits in R7. The EXPTB sub-routine cycles through normal, top-half double height and bottom-half double height.

The example application uses a single IR key (subroutine TIME) for both the display of the Teletext clock and the entry into sub-page mode. If the set is in TV mode then the time is displayed for 5 seconds. If the TV is in Text mode then sub-page mode is selected. Sub-page number entry is described in the following section. When the Teletext clock is requested it appears (boxed) at the top-right-hand corner. It is removed by the idle loop 5 seconds after the last press of the time button. When the time is being displayed all other Teletext activity is stopped using UCHOLD.

6g. Subroutines:  DIGITS, SUBPG, GET25 and GET26

DIGITS is the sub-page version of DIGIT0 and uses similar code. More checks on the input data are required as the four digits of the sub-page number have different maximum values. These maximums are 3 for thousands, 7 for the tens and 9 for the hundreds and units. These values reflect the sub-page number's original use as a time (24hr format). For tens and thousands a keyed 8 becomes a 0 and a 9 becomes a 1; for thousands only 4, 5, 6 and 7 become 0, 1, 2 and 3 respectively.

The code at the label SETIT is the sub-page equivalent of GETIT, described above. It requests the new sub-page and sets the FOUND flag.

The sub-routine SUBPG is called when the TIME (or clock) key is pressed (TV in Teletext mode). It toggles between normal mode and sub-page mode. When sub-page mode is entered the page number display (P—) is replaced with **** to indicate the mode change and to prompt for the entry of a sub-page number. Once all four digits have been entered the new sub-page is requested by SETIT. The code at the label RSTR is used to exit from this mode back to the normal (page number) mode, restoring the page number display to the top-left-hand corner.

GET25 is used by the idle loop to get the information stored in row 25 of the display chapter. This row is not displayed but contains various information used by the control microprocessor. The current page number, magazine number, sub-page number, Teletext control bits and the FOUND and PBLF flags are available. GET25 gets the required information and stores it in the RAM of the MC68HC05T7.

At the end of this sub-routine the I/O line 7, portB is checked. If it is low, packet 26 is handled. If it is high, this packet is disabled. This would be required if this application were to be used in a country other than Spain which used packet 26. It would require to be switched off as the enhanced display feature uses different characters depending on the country. In countries which do not use packet 26 (eg the UK) it does not matter whether or not packet 26 is enabled.

If packet 26 is enabled, GET26 processes all packet 26 data present in the ghost page. The tables G2TAB, G3TAB and CTAB contain the characters used to replace the character at the display location defined by each packet.
6h. Subroutines:  GIP, R24T and SR24T

The sub-routine GIP gets the initial (index) page from packet 8/30. It will be doing this as the set is brought out of standby or just after a channel change. It may thus initially get a poor signal (or there may be no Teletext) so it tries repeatedly until it finds a valid packet 8/30 format 1. If this is not found after 96 tries it gives up and sets the flag 6,STAT2 to indicate that there is no packet 8/30 (or no Teletext). In this circumstance it defaults to an index page number of 100.

R24T transfers bytes 10-30 of the broadcasting service data packet (8/30) into the display chapter. It is called once a second for five seconds after power-on or a channel change. The data is transferred to row 0 of the display page which can be displayed either at the bottom or, as in this example, the top of the screen. This transient display is setup using the sub-routine SR24T if Teletext is present. If the flag 6,STAT2 has been set by GIP as described above then SR24T does nothing. The transient display is terminated by code executed at the appropriate time from within the idle loop.

7. INPUT AND OUTPUTS

Apart from the IIC bus, only three pins on the controlling microprocessor are relevant to Teletext. Two inputs select the usage of packets 26 and 27 and one output can be used to control any hardware which requires to be changed according to whether or not there is a TV picture currently being displayed. In many applications some or all of these functions will not be required and could be eliminated from the software thus freeing up the pins for other uses.

PB3

This pin is active (high) during a pure (no-mixed, no-boxed) teletext display, otherwise it is low.

PB6

When this pin is low, Spanish use of link control bits 1, 2 and 3 is enabled. When it is high, these bits are ignored.

PB7

Packet 26 control. When low, packet 26 is enabled and handles all the Spanish alternate characters which are available in the 5243. When PB7 is high, packet 26 is ignored.

8. REFERENCES

Application note AN446, MCM2814 Gang-programmer using an MC68HC05B6.
LISTING 1

```
-rich text content-
```

190
139
140
***Updated page has arrived.***
141
***

145 000000ca >b600
146 000000cc >b700
147 000000ce >b900
148 000000d0 >b100
149 000000d2 4c
150 000000d3 0c0000
151 000000d6 >b0000
152 000000d9 a806
153 000000da >cd0000
154 000000dd a466
155 000000e0 >b700
156 000000e2 >b700
157 000000e4 a803
158 000000e6 >c50002
159 000000e9 a902
160 000000eb >b700
161 000000ed >cd0000

164 000000f0 a610
165 000000f2 >b700
166 000000f4 a906
167 000000f6 >b700
168 000000f8 >b700
169 000000fa >f000
170 000000fc >cd0000
171 000000ff 030c
172 00001000 >b400
173 0000100a >b700
174 00001010 >b100
175 00001015 >b500
176 0000101a 0100
177 0000101e 01
178 00001020 >b600
179 00001026 >b100
180 0000102c >b100
181 00001030 >c100
182 00001034 >d25
183 00001038 013b
184 0000103c 013b
185 00001040 >b600
186 00001044 >b600
187 00001048 >b600
188 0000104c >b600
189 00001050 >c0000

190 0000105f a023
191 00001061 b060
192 00001065 1b60
193 00001069 >b600
194 0000106b >b600
195 0000106f >b600
196 00001073 >b600
197 00001077 >b600
198 0000107b 1b80
199 0000107f 269
200 00001081 >b600
201 00001085 >b600
202 00001089 >b600
203 0000108d >b600
204 00001091 >b600
205 00001095 >b600
206 00001099 >b600
207 0000109d >b600
208 000010a1 >b600
209 000010a5 >b600
210 000010a9 >b600
211 000010ad >b600
212 000010af >b600
213 000010b3 >b600
214 000010b7 >b600
215 000010bb >b600
216 000010bc >b600
217 000010bd >b600
218 000010be >b600
219 000010bf >b600
220 000010c0 >b600
221 000010c4 >b600
222 000010c8 >b600
223 000010cc >b600
224 000010d0 >b600
225 000010d4 >b600
226 000010d8 >b600
227 000010dc >b600
228 000010e0 >b600
229 000010e4 >c0000

191
READ BSR READ1 GET FIRST BYTE
  LDA IOMBUF STA IOMBUF+1 MOVE IT UP
  CMP $A1 VM? INC SUBADR YES, NEXT SUB-ADDRESS
244 00000159 ad09 READ1 BSR IICSU
240 0000015d b600 BCLR $ADDR RM = 0 ALWAYS WRITE (SUB-ADDRESS)
246 0000015f ad00 LDA ADDR SEND CHIP-ADDRESS
242 00000161 b000 BSR SHIFT SEND SUB-ADDRESS
241 0000016b ad09 BSR SHIFT NO STOP BUT
240 0000016d b600 BCLR 5, MCR A RESTART
241 0000016f 163b BSET 5, MCR WAIT FOR IT
242 00000174 b69d BSET 0, ADDR SET BIT 0 FOR READ
243 00000176 b4d5 BSR ADDR RE-SEND CHIP ADDRESS
245 00000178 b13b BCLR 4, MCR CHANGE TO RECEIVER
246 0000017c b07b BCLR 3, MCR SWITCH OFF ACK.
248 0000017e 9a BCLR 7, MSR, WAIT FOR IT
249 00000181 f3c RETURN
250 00000184 81 MBINT CLR MSR
251 00000187 80 MBINT CLR MSR
LISTING 2

* TV/Teletext/OSD/Stereo program (MC68HC05T7).
* CCT Teletext control module (Spain).
* Used with RAMT8.S05, OST.S05 & TXT7.S05
* This software was developed by Motorola Ltd. for demonstration purposes.
* No liability can be accepted for its use in any specific application.
* Original software copyright Motorola - all rights reserved.

19th October '90

PORTA EQU $00 Port A address
PORTB EQU $01 Port B
PORTC EQU $02 Port C
PORTD EQU $03 Port D
PORTE EQU $04 Port E
PORTF EQU $05 Port F
PORTG EQU $06 Port G
PORTH EQU $07 Port H
PORTJ EQU $08 Port J
PORTK EQU $09 Port K
PORTL EQU $0A Port L
PORTM EQU $0B Port M
PORTN EQU $0C Port N
PORTO EQU $0D Port O
PORTP EQU $0E Port P
PORTQ EQU $0F Port Q

TIMERS

TCR EQU $12 Timer control register.
TSM EQU $13 Timer status register.
ICRH EQU $14 Input capture register, high.
ICRL EQU $15 Input capture register, low.
OCRH EQU $16 Output compare register, high.
OCRL EQU $17 Output compare register, low.
TDRH EQU $18 Timer data register, high.
TDLR EQU $19 Timer data register, low.
MISC EQU $1C Misc. register

OSC EQU $20 18 OSD data registers
CAS EQU $32 Color & status register
C34 EQU $33 Color 3/4 register
RAD EQU $34 Row address & character size
MCR EQU $35 Window/Column register
CCR EQU $36 Column/color register
HPD EQU $37 Horizontal position delay
MODR EQU $39 Test 1, OSD/Timer/PLM
MODR EQU $3A Test 2, EPROM
 **RAW allocation for Stereoton.**

**TEMPORARY MATRIX**

- SHA9MAT RMB 1
- TEMP RMB 1
- LUNK RMB 1
- SNDMD RMB 1
- ARAV RMB 1
- K1 RMB 1
- LVR RMB 1
- HVL RMB 1
- HVR RMB 1
- TONE RMB 1
- MATRIX RMB 1
- K2 RMB 1
- STA5 RMB 1
- STA6 RMB 1
- RMB 1
- RMB 1
- CAS1 RMB 1
- CCR1 RMB 1
- RAD1 RMB 1
- RAD2 RMB 1
- RAD3 RMB 1
- Rad4 RMB 1
- RA3 RMB 1
- RAD5 RMB 1
- RAD6 RMB 1
- RAD7 RMB 1
- Rad8 RMB 1
- CAS5 RMB 1
- CAS6 RMB 1
- CAS7 RMB 1
- CAS8 RMB 1
- CAS9 RMB 1
- CAS10 RMB 1
- CAS11 RMB 1
- CAS12 RMB 1
- DISL RMB 1
- LIN0 RMB 1
- NCHM RMB 1
- DIS0 RMB 1
- DIS1 RMB 1
- RMB 1
- ANE RMB 1
- TMF1 RMB 1
- TMF2 RMB 1
- RMB 12
- STACK RMB 22
- SP RMB 1
- DRAM RMB 128

**SECTION .RAM2.COMM**

**SECTION .ROM2**
DIGIT BCCLR 6,STAT, DIGIT

DIGIT BCCLR 3, R3

HOLD DURING

LDAX ACC

ENTRY

JSR UP

SENSOR 2, STAT

SET HOLD FLAG

LDA W2

SUB #16

LOG LDA POP

BNE NOCH NOT HUNDREDS SO DON'T CHANGE

CMN #1 YES, MORE THAN 7?

BLS NOCH NO, DON'T CHANGE

SUB #8 YES, 8->0 & 9->1

NOCH ADD #930 CONVERT TO ASCII

STA PAGE, X

UNITs?

CPX #2

HELP CLARFD YES, SO CLEAR PDP

LDA #920 DASH

CPX #1

TENS?

REQ TENS YES, SO LEAVE TENS

STA PAGE + 1 CLEAR TENS

TEN STA PAGE + 2 CLEAR UNITS

JBC PDP

BRA DCON

CLARFD CLR PDP

JMP STAT

CPX PDP

BSET 2, STAT

JSR SPM

JSR CNM

PAGE

PAGE HUNDREDS SO DON'T CHANGE

SAVE

PAGE HUNDREDS

ST.

IS PAGE ALREADY IN?

JSR SRCH

YES?

BNE INDEX DISPLAY CHAPTER

LDAX PAGE

PAGE HUNDREDS

STA PAGE, X

SAVE IN RAM

LDA PAGE + 1 PAGE TENS

LDA PAGE X1, X SAVE IN RAM

STA C O PAGE REQUEST TENS

LDA PAGE + 2 PAGE UNITS

LDA PAGE + 2, X SAVE IN RAM

STA CZ PAGE REQUEST UNITS

LDA PAGE PAGE HUNDREDS

SUB #18

STA R3 PAGE REQUEST HUNDREDS

LDA R4

JSR UP

JSR TDT REQUEST IT

LDA R3

JSR TDT REQUEST IT

JSR TDT REQUEST IT

LDA R3

BCLR 2, STAT

RESET HOLD FLAG

JMP SFND WRITE ONE TO FOUND

197
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PAGE JSR</td>
</tr>
<tr>
<td>0001</td>
<td>JP SET</td>
</tr>
<tr>
<td>0010</td>
<td>JP ON</td>
</tr>
<tr>
<td>0011</td>
<td>JP OFF</td>
</tr>
<tr>
<td>0100</td>
<td>JP IF</td>
</tr>
<tr>
<td>0101</td>
<td>JP IF NOT</td>
</tr>
<tr>
<td>0110</td>
<td>JP THEN</td>
</tr>
<tr>
<td>0111</td>
<td>JP THEN NOT</td>
</tr>
<tr>
<td>1000</td>
<td>JP HT</td>
</tr>
<tr>
<td>1001</td>
<td>JP HT NOT</td>
</tr>
<tr>
<td>1010</td>
<td>JP LQ</td>
</tr>
<tr>
<td>1011</td>
<td>JP LQ NOT</td>
</tr>
<tr>
<td>1100</td>
<td>JP HT ON</td>
</tr>
<tr>
<td>1101</td>
<td>JP HT OFF</td>
</tr>
<tr>
<td>1110</td>
<td>JP LQ ON</td>
</tr>
<tr>
<td>1111</td>
<td>JP LQ OFF</td>
</tr>
</tbody>
</table>

Note: The table above lists some common page and jump codes used in a specific context (e.g., a programming environment or system). Each code corresponds to a specific operation or condition that can be performed within the system's architecture.
634 ..............................
635 ..............................
636 ..............................
637 ..............................
638 ..............................
639 ..............................
640 ..............................
641 ..............................
642 ..............................
643 ..............................
644 ..............................
645 ..............................
646 ..............................
647 ..............................
648 ..............................
649 ..............................
650 ..............................
651 ..............................
652 ..............................
653 ..............................
654 ..............................
655 ..............................
656 ..............................
657 ..............................
658 ..............................
659 ..............................
660 ..............................
661 ..............................
662 ..............................
663 ..............................
664 ..............................
665 ..............................
666 ..............................
667 ..............................
668 ..............................
669 ..............................
670 ..............................
671 ..............................
672 ..............................
673 ..............................
674 ..............................
675 ..............................
676 ..............................
677 ..............................
678 ..............................
679 ..............................
680 ..............................
681 ..............................
682 ..............................
683 ..............................
684 ..............................
685 ..............................
686 ..............................
687 ..............................
688 ..............................
689 ..............................
690 ..............................
691 ..............................
692 ..............................
693 ..............................
694 ..............................
695 ..............................
696 ..............................
697 ..............................
698 ..............................
699 ..............................
700 ..............................
701 ..............................
702 ..............................
703 ..............................
704 ..............................
705 ..............................
706 ..............................
707 ..............................
708 ..............................
709 ..............................
710 ..............................
711 ..............................
712 ..............................
713 ..............................
714 ..............................
715 ..............................
716 ..............................
717 ..............................
718 ..............................
719 ..............................
720 ..............................
721 ..............................
722 ..............................
723 ..............................
724 ..............................
725 ..............................
726 ..............................
727 ..............................
728 ..............................
729 ..............................
730 ..............................
731 ..............................
732 ..............................
733 ..............................
734 ..............................
735 ..............................
736 ..............................
737 ..............................
738 ..............................
739 ..............................

- Read and write subroutines.
- Cyan 4 Index links & link control byte.
REVEAL $BSET 5,R7,REV
$BSET 5,R7
$BCLR 5,R7
$BRA OUT
$BCLR 4,R7,ROT
$BCLR 3,R7
$BRA OUT
SINGLE HEIGHT
$BSET 4,R7
$BRA OUT
BOTTOM
$BSET 3,R7
$BRA OUT
$BCLR 4,R7
TOP
$JMP TXT32
$TIME $BSET 4,STAT,HIGH
$TELETEXT CHIP?
$BCLR 4,STAT,CLOCK
$TELETEXT MODE?
$JMP SUBPG
$YES
$CLOCK $BSET 5,STAT,TAO
$NO, TIME Already ON?
LDA ACC
$STA R4
JSR VCHOLD
$BSET 4,STAT
$BSET 5,STAT
CLRA
BSR NOBX
LDA #$30
BSR BOXDOWN
JSR FR0
LDA #$09
STA R7
JSR TXT2
STOP FLASHES ON FIRST PRESS
LDA #$46
STA R5
STA R6
JSR TXT2
TAO $LDA #6
$STA TMH
RTS
NOS $STA R10
$LDA #$30
BSR BOX
LDA #$0B
BSR BOXDOWN
STA R10
LDA #$0A
BOX STA R11
LDA #6
LDA R4
LDA R8
CLR R9
LDA #6
$JMP TXT32
**Sub-page number entry routine.**

DEVICES JSR TPSTP
LDX W
SUB #1
S0D LDS PDP
REQ THOU
CPX THOU
THOU CMP #7
THOUSANDS OR TENS
 .'</>
BLX SOC
NO, DO NOT CHANGE
SUB #8
YES, 8->9 & 9->8
S0C TST# WAS CPX #0

DIGITS JSR TPSTP
LDx W
SUB #1
S0D LDS PDP
REQ THOU
CPX THOU
THOU CMP #7
THOUSANDS OR TENS
<table>
<thead>
<tr>
<th>Address</th>
<th>mnemonic</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000076c &gt;0002a</td>
<td>resets 6, STAT, OUTSP</td>
<td>-</td>
</tr>
<tr>
<td>000076f &gt;0000</td>
<td>resets 6, STAT</td>
<td>-</td>
</tr>
<tr>
<td>0000771 ad8a</td>
<td>resets TRSP</td>
<td>-</td>
</tr>
<tr>
<td>0000773 &gt;0000</td>
<td>clr PDP</td>
<td>-</td>
</tr>
<tr>
<td>0000775 &gt;0000</td>
<td>jsr INOX</td>
<td>-</td>
</tr>
<tr>
<td>0000778 &gt;0000</td>
<td>lda PAG, X</td>
<td>-</td>
</tr>
<tr>
<td>000077a &gt;0000</td>
<td>sta PAGE</td>
<td>-</td>
</tr>
<tr>
<td>000077c &gt;0000</td>
<td>lda PAG+1, X</td>
<td>-</td>
</tr>
<tr>
<td>000077e &gt;0000</td>
<td>lda PAG+2, X</td>
<td>-</td>
</tr>
<tr>
<td>0000780 &gt;0000</td>
<td>sta PAG2</td>
<td>-</td>
</tr>
<tr>
<td>0000784 ad2a</td>
<td>lda #2</td>
<td>-</td>
</tr>
<tr>
<td>0000786 &gt;0000</td>
<td>sta R1</td>
<td>-</td>
</tr>
<tr>
<td>0000788 &gt;0000</td>
<td>sta PH</td>
<td>-</td>
</tr>
<tr>
<td>000078a &gt;0000</td>
<td>sta PT</td>
<td>-</td>
</tr>
<tr>
<td>000078c &gt;0000</td>
<td>lda ACC</td>
<td>-</td>
</tr>
<tr>
<td>0000790 &gt;0000</td>
<td>sta R8</td>
<td>-</td>
</tr>
<tr>
<td>0000792 ad02</td>
<td>lda #2</td>
<td>-</td>
</tr>
<tr>
<td>0000794 ad02</td>
<td>sta R10</td>
<td>-</td>
</tr>
<tr>
<td>0000798 ad02</td>
<td>brsr TXT3</td>
<td>-</td>
</tr>
<tr>
<td>000079a &gt;0000</td>
<td>jmp TRAP1</td>
<td>-</td>
</tr>
<tr>
<td>000079d ad00</td>
<td>OUTSP BRTR</td>
<td>-</td>
</tr>
<tr>
<td>000079f &gt;0000</td>
<td>lda ACC</td>
<td>-</td>
</tr>
<tr>
<td>00007a1 &gt;0000</td>
<td>jsr UP</td>
<td>-</td>
</tr>
<tr>
<td>00007a4 &gt;0000</td>
<td>brcl 2, STAT</td>
<td>-</td>
</tr>
<tr>
<td>00007a6 &gt;0000</td>
<td>jsr TXT1</td>
<td>-</td>
</tr>
<tr>
<td>00007a9 &gt;0000</td>
<td>jmp TXT3</td>
<td>-</td>
</tr>
</tbody>
</table>

* Read in Row 25 information. *
1572
1573  * Set up Row 24 transient.
1574
1575
1576
1577 1578 0000ae 9c 0002 SET 6,STAT,NOTXTX
1579 0000aad 1a00 SR24T BSET 5,STAT "TIME HOLD"
1580 0000aad 4f CLRA
1581 0000aad c0000 JSR UP . ACC 0
1582 0000af1 1700 BCLI 3,83 STOP IT
1583 0000af3 a604 LDA h 3 BYTES
1584 0000af5 c0000 JSR SPM
1585 000000 1a00 LDA h
1586 0000af6 b700 STA R4
1587 0000af6 a606 LDA h
1588 0000af7 b700 STA R5
1589 0000af7 b700 STA R6
1590 000000b2 a684 LDA h #84 PUT 24 AT TOP ($44 FOR CURSOR)
1591 000000b4 b700 STA R7
1592 000000b6 c0000 JSR TXT2
1593 000000b9 a606 LDA h
1594 000000bd b700 STA TM
1595 000000be b100 NOTXTX BSET 4,STAT
1596 000000bf 81 RTS
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607 000080 h 30313233343353637 NUM FCC "0123456789ABCDEF"
1608 000080 h 300003443500a0a FIELD PCB 508,508,508,508,508,508,508,508
1609 000080 h 150249a5463382f hAM PCB 515,502,549,556,564,573,538,52f
1610 000080 h dc78c90a1h4dea HAM PCB 500,507,18c,598,161,186,57d,1ca
1611 1612 END

Symbol cross-reference

.RAM  *27
.RAM2  *27
.BOM2  *29
.AV  *27
.ABC  216 220 222 229 261 263 265 267
.ARDF  614 623
.ADF  *93
.ART  *54
.ASCII  469 502 506
.ACC  *27 54 106 179 181 205 217 223 231 245 288 309 316 317 318
.ADF  363 350 561 586 593 616 846 861 864 868 915 929 945 981 1053
.ADCON  *27 662 1120
.ADF  *34 353
.AXDF  *27
.AXAL  *27
.AVOL  *27
.BC  593 599
.BLANK  968 971
.BIT  1003 1006 1009
.BX  1003 1006 1009
.BXDCF  32 822 1007
.BXCON  22 826 989 1004 1057
.BXI  *27
.BB  27
.BBON  27
.BW  27 165 276 483 503 939 1086 1179
.BX  27 164 273 475 677 840 500 962 1088 1182
.C  27 878 1090 1221 1224
.C  27 879 1092 1216 1219
.CX  27 880 1094 1239 1242
.C6  *27 242 882 1696 1204 1207
.CARO  234 238
.CAS1  *27
.CAS2  *27
.CAS3  *27
.CAS4  *27
.CAS5  *27
.CAS6  *27
.CAS7  *27
.CAS8  *27
.CCR1  *27
.CCR2  *27
.CCR3  *27
.CCR4  *27
.CCR5  *27
.CCR6  *27
.CCR7  *27
.CSI  1359 1361
.CPRG  246 613
.CGRT26  1240 1264
.CH  *27
.CHC1  548 559
.CHC2  *540 543
.CHFD  1339 1347
.CHDF  1345 1350 1360 1402
.CLINK  21 676
.CLOCK  978 980
.CLARO  121 129 204 214

---

Set up Row 24 transient.

Tables for HEX-ASCII conversion, "STOP" and Hamming decode.
Symbol cross-reference

CNT1  *27
CNT3  *27
CNT4  *27
CNT5  *27
CNT6  *27
CNT7  *27
CNF  367  *369
COUNT  *27  312  320  325  332  339  341  342  349  353
CPBF  613  *945
CTAB  1337  1366  *1432
CYAN  17  *241
CVOU  *297
DDI  *574
DECODE  372  375  404  407  572  690  *700
DIFFX  *27
DIFFL  *27
DIGIF  103  *105
DIGITD  17  *103
DIGITD9  104  *1023
DISC  *27
DISP  *27
DISP4  893  *898
DISP4  897  *898
DPGR  128  *130
DPRT  *27
DRAM  *27
EA  *1151  1559
END24  1246  1375  *1409
EXAD  1310  *1547
EXP  967  *973
EXPTR  18  *967
FI  758  760  *761
FINI  536  539  *543
FND  721  730  739  754  763  772  781  785  *787
FND2  542  *548
FND3  704  712  *730
FO1  749  751  *752
FRD  828  *842  990
FTUNE  *27
G3YF  1384  *1398
G2BIT  1371  *1388
G3TAB  1390  *1398  *1417
G3IF  1381  *1385
GBIT  *1375
G2TAB  1377  1385  *1429
GET  336  *665
GET24  1244  *1253
GETIND  *1493  1513
GETIT  *158
GIF  18  259  *1460  1570
GLOR  209  *215
GLP1  323  *365  667  672
GLP2  352  *471  669  674  1503
GOTCH  1367  1384  1399  *1403
GREEN  17  *208
GT  1352  *1363
HI  385  *388
HANG  702  *714  723  732  747  756  765  776  *1609
HANH  *1610
HIST  952  *954  977
HOLD  19  *958
HUN  1244  *1048
HYL  *27
IAC  218  *266  *269
ID0  215  *217
ID1  221  *223
ID2  262  *264
INDEX  19  *256
INOX  160  *179  281  379  424  618  931  1141  1172
INP  201  213  *618
INPT  19  *170
IOBUF  *27  371  374  384  390  395  403  406  571  573  595  597  683  689  952  1203
IPNF  1479  *1504
IACMCT  *27
IACMT  *27
IACODE  *27
IAG  *27
IAG  *27
IAR1  *27
IAR2  *27
IAR3  *27
IAR4  *27
IAX  *27
IAX  *27
IAX  *27
IAX  *27
IAX  *27
IAX  *27
IAX  *27
L10  *113
L22  899  901  903  905  *1608
LIFO  *27  518  626  1254  1256  1256  1260  1262  1263  1267  1270  1272  1277  1279  1279  1288  1292
L32  1399  1399  1299  1299  1304  1305  1309  1311  1320  1323  1327  1330  1331  1336  1338  1347
L33  1349  1362  1364  1369  1375  1380  1388  1393  1401  1402  1447  1448  1449  1450  1451  1452
LIND  *27
LINK  21  *309
LLOOP  *341  355
LLOP  *320  334
LAX  *505  *508
LOOP16  *1256  1271  1290
LOOP22  *1276  1321  1379  1392  1407
LOOPS  *530  346
LPT  203  206  213  218  *224
LPT2  159  *204
LVR  *27
LVR  *27

212
Symbol cross-reference

<table>
<thead>
<tr>
<th>Symbol</th>
<th>V5</th>
<th>W1</th>
<th>W2</th>
<th>W2B</th>
<th>W3</th>
<th>WACC</th>
<th>WBCW</th>
<th>YELLOW</th>
<th>YIP</th>
<th>ZEI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>268</td>
<td>27</td>
<td>656</td>
<td>700</td>
<td>703</td>
<td>791</td>
<td>835</td>
<td>884</td>
<td>1114</td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>111</td>
<td>373</td>
<td>377</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>602</td>
<td>641</td>
<td>1289</td>
<td>1554</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>224</td>
<td>226</td>
<td>230</td>
<td>243</td>
<td>314</td>
<td>321</td>
<td>329</td>
<td>331</td>
<td>890</td>
</tr>
<tr>
<td></td>
<td></td>
<td>224</td>
<td>226</td>
<td>230</td>
<td>243</td>
<td>314</td>
<td>321</td>
<td>329</td>
<td>331</td>
<td>890</td>
</tr>
<tr>
<td></td>
<td>1556</td>
<td>1557</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1477</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>337</td>
<td>347</td>
<td>494</td>
<td>529</td>
<td>530</td>
<td>532</td>
<td>543</td>
<td>544</td>
<td>548</td>
</tr>
<tr>
<td></td>
<td></td>
<td>220</td>
<td></td>
<td>1183</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>558</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>560</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1501</td>
</tr>
<tr>
<td></td>
<td>707</td>
<td>709</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INTRODUCTION

The MC68HC11K4 is provided with memory expansion logic which allows the 64K Byte addressing range of the 68HC11 CPU to be extended to more than 1 Mbyte. This application note discusses the operation of this logic and provides examples of memory maps and possible hardware configurations.

THE MC68HC11K4 MEMORY EXPANSION LOGIC

The memory expansion logic extends the addressing range of the 68HC11 CPU by providing two new on-chip blocks. The first new block implements additional address lines which are only made active when required by the CPU. The second block eases interfacing to external memory chips by providing chip select signals. Both of these blocks are fully user programmable.

ADDITIONAL ADDRESS CAPABILITY

If the addressing capability of the 68HC11 CPU is to be extended then the first step involved is to provide additional address lines. The CPU itself provides 16 lines (A0 to 15) which allow up to 64K Bytes of memory to be accessed. Each new address line provided will double that total.

To maintain compatibility with other members of the 68HC11 family the CPU used in the K4 is not functionally changed. The extra addressing capability is provided in such a way that the CPU itself is unable to distinguish more than 64K Bytes of memory. The extra capacity is provided by switching banks of the extra memory in and out of the 64K Byte range provided. To maximise the flexibility of this approach the size and number of the switched banks is user programmable.

To use extended memory, the programmer must first allocate a range (called a window) within the CPU 64K Byte addressing range which will be used when banks are switched in and out. The memory expansion logic allows windows of 8K, 16K or 32K Bytes to be defined and placed at programmable points in the CPU 64K Byte memory. At any time only one bank may be displayed in the defined window and therefore the CPU may only have access to the memory contents of one bank at a time. The bank which is displayed in the window is selected by the additional address lines provided by the memory expansion logic. The process of replacing the active bank with a new bank is called bank-switching.

A useful analogy is that of photographic transparencies and a projector. The viewer may have many transparencies but is only able to view one at a time in the projector. If the photographs are numbered then the viewer is able to select precisely which one to view without having to go through all of them. Here the memory banks are akin to the transparencies - many are available but only one is accessible at any time. The number on the transparency can be thought of here as an address. Any transparency which is not being displayed at any one time is still accessible but only when the current one is removed and it is put in its place.
To extend the addressing capability of the CPU, six address lines were added. These are termed XA13, XA14, XA15, XA16, XA17 and XA18. To allow flexibility in the size of the windows, the lower three address lines are only used when determined by the size of the window and replace the CPU's equivalent addresses.

To understand the need for the XA13 to XA15 addresses consider the case when an 8K Byte window is being used. Address lines XA16 to XA18 are only active when the CPU is accessing memory within the window and they provide an extra 8 \(2^3\) times the memory provided by the CPU within that window. If an 8K Byte window is used then a maximum of 8 x 8K Bytes is available. However, for the CPU to uniquely distinguish each location in the 8K Byte window it only requires to use addresses A0 to A12. Changes in address lines A13 to A15 take it outside of the window and are therefore never valid within the window except to identify the starting address of the window. The three new addresses XA13 to XA15 are provided so that a full 512K Byte (8 x 8 x 8K Byte) range is realised. For a 16K Byte window only addresses XA14 and XA15 can be used and for 32K Byte only XA15 is usable. Note that the size of the memory window to be used need not necessarily be defined at the hardware design stage since the additional addresses XA13 to XA15 can be programmed to carry the CPU A13 to A15 signals. The situation may be complicated by the need to use differently sized windows (see example 2).

The memory expansion logic actually allows the user to define two independent windows and so more than 1M Byte of memory is accessible.

The hardware required to implement such a large memory range is greatly simplified by the use of the memory expansion's chip select block.

CHIP SELECTS

The memory expansion chip selects are provided to help the user interface the K4 with external memories. Four are provided but only three are of direct importance to the memory expansion logic. These are the two General Purpose Chip Selects and the Program Chip Select. The fourth, I/O Chip Select, is used to simplify the addition of external peripheral chips.

The basic function of a chip select is to provide a logic signal to indicate that the CPU is accessing a certain area of memory. For example, the Program Chip Select is intended to be active in the range of memory where the main program exists. Other chip selects will be active when their respective memory areas are used.

The General Purpose Chip Selects are the most flexible of those provided and their function is closely linked to the memory expansion logic. They can be programmed to be active on an area either within the CPU 64K Byte memory or within either window's 512K Byte range. In both cases the size of memory selected is fully programmable from 2K Bytes to 512K Bytes.

The above paragraphs outline the method by which the memory expansion logic extends the addressing range of the CPU. A detailed description of the internal registers used to implement the new logic is now required. Finally a series of examples are considered.

MEMORY EXPANSION AND CHIP SELECT REGISTERS

This discussion describes the functionality of the internal registers relating to the memory expansion logic and chip selects. Further details on their addresses and specific bit operations may be found in the Technical Summary [1].

The following registers perform the memory expansion function.

The MMWBR register allows the starting address of each of the two windows within the CPU 64K Byte address range to be defined. The windows will normally start on a boundary related to their size, for example an 8K Byte window may start on any 8K Byte boundary starting at $0000$, that is, $2000$ $4000$ ... $E000$. A 16K Byte window can only start on 16K Byte boundaries, $0000$ $4000$ ... $C000$. An exception is made for the 32K Byte window. This would normally start at either $0000$ or $8000$. However, the window is also allowed to start at $4000$.

The MMSIZ register sets the size of the windows in use and selects whether the chip selects are active for only CPU addresses or for extended addresses.
Each General Purpose Chip Select has two registers called GP1CSC, GP1CSA and GP2CSC and GP2CSA.

The control register (GP7CSC) determines the logical output required when an area of memory is selected (with possible logic combinations with other chip selects) and the range of memory over which the chip select is to be active. Each chip select can be programmed to become active whenever the CPU address enters an memory expansion window (regardless of the actual bank selected); this is known as following a window.

The address register (GP7CSA) allows the starting address of the chip select to be programmed. The bits in this register which are active are determined by the size of the chip select range selected by the control register.

The program and I/O chip selects are programmable via the CSCTL register.

Two window registers, MM1CR and MM2CR, are used to indicate which bank is active in a window. Each contains the values of XA13 to XA18 to be output when the CPU selects addresses within the extended memory window. To change banks the user writes the address of the new bank into the appropriate window register.

The actual memory expansion address lines are multiplexed with PORT G I/O pins. Selecting an address line on one of these pins means that a PORT G pin is lost. For this reason the user need only select those address lines which are needed by the expansion logic. This allows unused lines to be used as general purpose I/O. The register which defines which extended address lines are used is PGAR. If an address line is not required then the appropriate bit in PGAR should be cleared to 0. A special case exists for two address lines which overlap the CPU address lines (XA13 and XA14). If XA13 or XA14 are selected as address lines in PGAR, but are not used in either window, then the appropriate CPU address line will be output on the port.

EXTENDED MEMORY EXAMPLES

The best way to grasp the implications of the K4 extended memory function is to consider some examples. Each example consists of two figures. Figures a are the logical arrangement of the memory and Figures b are a possible hardware configuration.

Example 1 shows one window in use. This is an 8K Byte window scheme and provides 8 banks from a single 64K Byte EPROM chip. Note that the logical address of each bank is derived from address lines A0 to A12 then XA13 to XA15. Chip select 1 is used.

Example 2 shows two windows in use. The first window is of 8K Bytes and is organised as in example 1. The second window is of 16K Bytes and is organised as 16 banks in two 128K Byte RAM chips. The logical addresses of the Window 2 banks are determined by A0 to A13 then XA14 to XA17 (XA18 determines start address). Chip select 2 is used for window 2.

Example 3 shows the same two windows as in example 2 except that the logical addressing of the windows are changed. Now Window 1's logical address is determined by A0 to A12 then XA15 to XA17 (XA13 and XA14 ignored) and Window 2's logical address is determined by A0 to A13 then XA15 to XA18 (XA14 ignored). Note that in both windows every bank will be duplicated due to the lack of decoding on certain address lines. In Window 1 each bank is duplicated four times. In Window 2 each bank is duplicated twice.

Example 4 shows the maximum 1Mbyte extended memory possibility in use. Window 1 is 16K Bytes starting at $0000 and Window 2 is 32K Bytes starting at $4000. Note that the internal RAM and registers of the K4 are echoed in every page of window 1, but that the internal EPROM in window 2 only occurs in the first 64K Bytes of extended memory. That is, for addresses below $10000, the internal EPROM is present in the memory map at the relevant address. This currently applies to all window sizes and configurations, however, the user should avoid referring to EPROM at any address beyond page 0 to ensure compatibility with any future upgrades.

Logical addresses of both windows are given by A0 to A13 then XA14 to XA18. Window 2 uses XA14 because it does not start on a 32K Byte boundary and so requires that the XA14 is inverted. Both chip selects are used and follow a window each.
PGAR $3F - XA13..XA18  MMSIZ $42 - Window 1 8K
MMWBR $04 - window at $4000  CSCTL $00 - no I/O or program chip select
GPCS1A $00 - from $00000  GPCS1C $06 - 64K range (8 x 8K)
GPCS2A $00 - not relevant  GPCS2C $00 - disabled

Figure 1a. One 8K Byte window
Figure 1b. 64K Bytes in one 8K Byte window
Figure 2a. One 8K Byte window and one 16K Byte window

PGAR $3F - XA13..XA18  MMSIZ $E2 - Window 1 8K, Window 2 16K
MMWBR $84 - Window 1 at $4000, Window 2 at $8000  CSCTL $00 - no I/O or program chip select
GPCS1A $00 - from $000000  GPCS1C $06 - 64K range (8 x 8K)
GPCS2A $80 - from $400000  GPCS2C $08 - 256K range (16 x 16K)
PGAR $3C - XA15..XA18  MMSIZ $E2 - Window 1 8K, Window 2 16K
MMWBR $84 - Window 1 at $4000, Window 2 at $8000  CSCTL $00 - no I/O or program chip select
GPCS1A $00 - from $00000  GPCS1C $08 - 256K range
GPCS2A $00 - from $00000  GPCS2C $09 - 512K range

Figure 3a. 8K Byte window and 16K Byte window with new addressing
PGAR $3E - XA14..XA18  
MMSIZ $F2 - Window 1 16K, Window 2 32K
MMWBR $40 - one window at $0000, one at $4000  
CSCTL $00 - no I/O or program chip select
GPCS1A $00 - not relevant  
GPCS1C $0A - follow window 1
GPCS2A $00 - not relevant  
GPCS2C $0B - follow window 2

Figure 4a. Possible 1MByte addressing organisation
Figure 4b. One MByte in two windows
CONCLUSION

The standard 64K Byte addressing range of the M68HC11 family is easily extended using the MC68HC(7)11K4 memory expansion. This logic also provides programmable chip selects to allow easy interfacing with external memory chips. A similar extended memory system may be implemented on other M68HC11 products by following other systems as described in [2].

REFERENCES

[1] MC68HC711K4 Technical Summary, Reference BR751/D
[2] "128K byte addressing with the M68HC11", Reference AN432/D
INTRODUCTION

Development systems for single-chip MCUs can be complex and expensive. This can dissuade potential users of this type of microprocessor from designing them into new applications. This application note describes a simple "entry" development system suitable for debugging hardware and software for the M6805 range of microprocessors. The M6805 range includes both CMOS and NMOS parts, most of which are single-chip devices which include mask-programmable ROM, RAM, I/O and one or more timers. The exceptions are the CMOS MC146805E2 and MC68HC05EO which have no on-chip ROM but instead have data and address buses which enables them to use external memories and peripherals.

The development system uses the MC68HC05EO processor and can be used to develop applications intended for any M6805 but is particularly suitable for applications which will use the MC68HC05EO itself. The MC68HC05EO has a non-multiplexed bus which requires no additional hardware to interface with RAMs, ROMs and EPROMS. It has 480 bytes of RAM, 3 timers, 3 chip-select lines and 4 8-bit ports enabling it to meet many requirements with the addition of only an EPROM containing the application software.

Such a system will be most cost effective in small volume applications not justifying a mask programmed single-chip MCU and also in applications requiring larger programs than can be accommodated in ROMed MCU versions.

For software development, however, RAM and a monitor have to be incorporated so software can be loaded and de-bugged. An MCM6264 8Kbyte (or MCM60L256/MCM6206 32Kbyte) RAM is used. The EBUG05 monitor EPROM, listed at the end of this application note, fulfils the monitor requirement.

In an E0 application the development system can be used as an add-on to the target system hardware, as shown in Figure 1 (components to the left of the dotted line). The application hardware needs very little modification as it already contains the MPU, the interface to the development system being mostly via the socket for the EPROM. Apart from the keyboard and display connections which use port A, only four or five additional connections are required. These can be made available on a small connector on the application PCB. In one of the 8K applications, used to check out the development system (reference 1), a keyboard and display were required, so only four I/O lines (see below) were dedicated to the development system. In a 16K application, A13 (PD5) is also required, so a fifth I/O line would be used. The keyboard provides up to 32 keys using 7 I/O lines and an external 3-line to 8-line decoder. One additional I/O line is used for the display. If the keyboard and/or display is not required in the final application (e.g., reference 2) then the I/O lines, used by them during development, are available for other purposes but their use cannot readily be de-bugged. This is a disadvantage of such a simple development system. There is, however, a major advantage of this system compared with those which use the display and keyboard of a PC. In this system the PC can be used to display the program listing file, obviating the need for printouts during debug.
CIRCUIT

Figure 1 shows the circuit used. The 8K or 16K (half of an MCM60L256) RAM is placed between $0000$ and $3FFF$; this means that the 512 bytes from $0000$ to $01FF$ are not used, as they are mapped over the EO's I/O and RAM space. Locations $0200$ to $021F$ are used by the monitor, so use of the RAM should start at $0220$ or above. Some RAM may be required to replace EO page 0 RAM (16 bytes), used by the monitor or for other purposes during de-bug; the application used to check out the monitor used RAM from $0400$. This provides 7K (15K using an MCM60L256) of RAM for the code being de-bugged. The start address chosen for the code being developed can be extended down to $0220$ if this RAM is not required, providing a program space of nearly 15/2Kbytes. The 16K limitation is caused by the split of the memory map into four (see Figure 3), to simplify address decoding and facilitate the ability to load code from an EPROM. If this capability is not required, more complex address decoding would allow up to 60K of RAM to be available for code; the monitor uses less than 4K.

The MC74HC138 provides the chip selects during de-bug as the on-chip chip selects signals are not suitable for the monitor's address map. The final system will usually not require an MC74HC138.

The 28-pin EPROM socket which will contain the final de-bugged code is used during development as the main interface to the monitor and its RAM. Only four or five other signals are required. These are the two or three most significant address lines, A15, A14 and, optionally, A13 and the clock (P02), which are required by the MC74HC138, the R/W line and a port line (P02) for inputting or outputting code on an RS232 serial link. PB2 can be used in the application as it is only used by the monitor during serial transfers when the application is not running. Care should, however, be taken to avoid contention within the hardware connected to PB2. If only 8K bytes of external RAM are required, then A13 is not required and its pin, PD5, can be used by the application.

The hardware connected to the EPROM socket can also include a 28-pin socket to accommodate a 27(C)128 EPROM. This is addressed between $4000$ and $7FFF$ and can be used to introduce software from an EPROM which has been programmed with code for de-bug. The monitor contains a routine which transfers the contents of this EPROM into RAM. Alternatively, code can be loaded serially via the RS232 interface.

The EBUG05 monitor EPROM (27C64) included in the external hardware is enabled from $C000$ to $FFFF$, although its actual start address is $E000$. Figure 3 shows the memory map of the development system.

The monitor display is a 6-digit 4-backplane LCD (e.g., Hamlin type 4200 or the 8-digit GE type LXD69D3F09KG), which is driven by an MC145000P display driver. The driver is controlled by a 2-line serial link from the microprocessor. A single-backplane display can be used as an alternative, as shown in Figure 2. Three MC144115 driver chips are used along with a transistor inverter to supply the additional latch enable signal required by these drivers. This circuit requires more connections to the LCD but allows the use of a more commonly available display.

The optional RS232 interface can be implemented using the single-supply MC145407 driver-receiver chip. If outputting of S-records is not required, then a simple transistor inverter with a pull-up resistor and a reverse polarity protection diode can be used. This interface is shown in Figure 4.
Figure 1. MC68HC05E-based M6805 development system
Figure 2. Alternative static LCD display

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>MC68HC05E0I/O, timers</td>
</tr>
<tr>
<td>001F</td>
<td>MC68HC05E0 RAM</td>
</tr>
<tr>
<td>0020</td>
<td>(16 bytes used by EBUG05)</td>
</tr>
<tr>
<td>002F</td>
<td>MC68HC05E0 RAM</td>
</tr>
<tr>
<td>0030</td>
<td>(464 bytes including stack at 00FF), for use in application</td>
</tr>
<tr>
<td>01FF</td>
<td>MC68HC05E0 RAM</td>
</tr>
<tr>
<td>0200</td>
<td>(32 bytes used by EBUG05)</td>
</tr>
<tr>
<td>021F</td>
<td>External RAM,*</td>
</tr>
<tr>
<td>0220</td>
<td>can be used for data and/or program</td>
</tr>
<tr>
<td>3FFF</td>
<td>Optional EPROM socket</td>
</tr>
<tr>
<td>4000</td>
<td>not used</td>
</tr>
<tr>
<td>7FFF</td>
<td>EBUG05 monitor</td>
</tr>
<tr>
<td>8000</td>
<td></td>
</tr>
<tr>
<td>BFFF</td>
<td></td>
</tr>
<tr>
<td>C000</td>
<td></td>
</tr>
<tr>
<td>FFFF</td>
<td></td>
</tr>
<tr>
<td>FFFF</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Memory Map
It is often useful with CMOS circuits to provide a simple IDD monitor which shows via an LED whether or not the IDD is above or below a specific value. In this application it shows whether or not the microprocessor is in the STOP mode. The required circuit is shown in Figure 5. The current threshold can be chosen by selecting the value of R1. A value of 1kohm sets the limit at about 500μA which means the LED should be off when the E0 is in STOP mode and on otherwise. The 500μA limit allows the LCD and perhaps a CMOS target application to be supplied without switching on the LED. When the microprocessor is not in STOP mode, its IDD is several milliamps and the LED should be lit. If a multiplexed LCD is used, it may be preferable not to supply it via this type of monitor circuit, as a significant change in contrast may occur as the microprocessor goes into its STOP mode (see Figure 5). The monitor drops about 600mV when the microprocessor is running, so the supply voltage should be chosen accordingly. While the microprocessor will operate down to 3V (at an appropriate frequency), the RAM and EPROM chips are specified at 5V±0.5V. For battery operation 4 zinc-carbon or 5 Ni-Cad cells can be used.
EBUG05 KEY FUNCTIONS

The EBUG05 EPROM comprises a monitor, developed from the MC146805E2 CBUG05 program, specifically written to enter and debug 6805 code. The following functions are available using the 24-key keyboard:

<table>
<thead>
<tr>
<th>Function</th>
<th>Key</th>
<th>Description of function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>0</td>
<td>Display program counter.</td>
</tr>
<tr>
<td>BP</td>
<td>5</td>
<td>Enable breakpoints. The first breakpoint is displayed if enabled or off if it isn't. Enter new breakpoint address followed by ENTER to change, or just ENABLE to move to next one, three breakpoints are available. This number can be increased, the only cost being the additional RAM used (3 bytes per breakpoint).</td>
</tr>
<tr>
<td>BPC</td>
<td>6</td>
<td>Clear breakpoints. ENTER clears all breakpoints. Entering a number followed by ENTER clears that breakpoint only.</td>
</tr>
<tr>
<td>OFF</td>
<td>A</td>
<td>Calculate branch offset. The address of the branch instruction and that of the destination are requested. If a valid branch is calculated, it is written into memory and displayed. If not valid, then “or” for out of range is displayed. A branch of -128 through +127 relative to the start address of the next instruction is allowed.</td>
</tr>
<tr>
<td>TR</td>
<td>B</td>
<td>“Trace” one instruction. This is not a true trace as breakpoints are advanced sequentially through the code and do not follow branches or jumps (see below).</td>
</tr>
<tr>
<td>STOP</td>
<td>C</td>
<td>Put the E0 into STOP mode, clock stops.</td>
</tr>
<tr>
<td>WAIT</td>
<td>3</td>
<td>Put the E0 into WAIT mode.</td>
</tr>
<tr>
<td>CC</td>
<td>D</td>
<td>Display/change Condition Code register, new data is followed by ENTER. ESC returns to “□” prompt without changing contents.</td>
</tr>
<tr>
<td>XR</td>
<td>E</td>
<td>Display/change index Register, new data is followed by ENTER. ESC returns to “□” prompt without changing contents.</td>
</tr>
<tr>
<td>AR</td>
<td>F</td>
<td>Display/change Accumulator, new data is followed by ENTER. ESC returns to “□” prompt without changing contents.</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>Output S-records. When this key is pressed, “bA” for begin address is displayed; enter first address (and ENTER), then last address when “EA” is displayed. Another ENTER starts the RS232 S-record output.</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>Load S-records from the RS232 interface. Press L and “LOAD” is displayed. S-records sent to 2, PORTB, will be loaded until an S9 record is received. The prompt returns if the load was OK. If not, an error message is displayed (see next section).</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
<td>Verify RAM against S-records. “VERIFY” is displayed, otherwise the same as LOAD including error checks.</td>
</tr>
<tr>
<td>ENTER</td>
<td>ENT</td>
<td>Enter keyed-in address or data (and move to next address in “M”).</td>
</tr>
<tr>
<td>ESC</td>
<td>ESC</td>
<td>Escape from current function (except STOP, WAIT, V, L &amp; P).</td>
</tr>
</tbody>
</table>
### Function Key Description of function

<table>
<thead>
<tr>
<th>Function</th>
<th>Key</th>
<th>Description of function</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO</td>
<td>GO</td>
<td>Begin or continue program. When pressed current PC is displayed. To proceed from that location, press ENTER; to proceed from a different address, enter the address followed by ENTER.</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>Display/change a location in RAM. When pressed, last address is displayed. Press ENTER to display the contents of this address or enter a new address, followed by ENTER. New contents can be entered (followed by ENTER) if required. ENTER moves to next address; M moves to previous address.</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
<td>Display stack pointer; cannot be modified.</td>
</tr>
<tr>
<td>XFER</td>
<td>7</td>
<td>Transfer code from an EPROM between $4220 and $7FFF into RAM ($0220 to $3FFF). When pressed, default start address (in RAM) is displayed. Press ENTER to start at this address ($0400), or enter new start address followed by ENTER. End address (in RAM) is displayed. Press ENTER to transfer code up to this end address ($1FFF), or enter new end address followed by ENTER. The lowest allowable start address is $0220 and the highest end address is $3FFF. During the transfer the code is read from the EPROM at the RAM addresses plus $4000. The code in EPROM should have been assembled with the start address where it will reside during execution (e.g., $0400).</td>
</tr>
</tbody>
</table>

### USE OF THE MONITOR

The MC68HC05E0's vectors are within the address space of the EBUG05 EPROM. They operate via extended jumps in RAM. This gives the user access to the vectors if these jumps are written to, from within the user's program. The vectors' locations are shown in Table 1. The extended jump instruction ($CC) is written to RAM by the monitor; all the user has to do is to overwrite the destination addresses at the locations indicated in the Table. This must be done at the start of the user code, before interrupts are enabled. Vectoring the jumps through a table at a fixed location in the user code obviates the need to change this initialisation when re-assembling changes the addresses of the interrupt service routines. Lines of code for this purpose are included as comments in reference 2.

Clearly the RESET vector cannot be altered in this way, so during debug user software should be started using the GO facility in EBUG05. Software interrupts (SWIs) are used by the monitor to facilitate breakpoint and should therefore not be used in the application software.

Software for de-bug should be assembled to reside in RAM, starting at address $400 (or down to $220 if more space is required, see above). It can be introduced into the system in an EPROM at $4000 or loaded serially from a PC via the RS232 interface. The monitor includes a routine to move the code from EPROM to RAM, where it can be executed and debugged using EBUG05. This provides an alternative method of loading code for debug if a serial load is not appropriate. When debug is complete, the code should be re-assembled at $E000 (or $C000 if a 16K EPROM is used). When the code is in EPROM, the vectors should be included. The jumps required during debug are then no longer relevant.

CSROM will normally be suitable for selecting this EPROM, so that the MC74HC138 will not be required once the code has been finalised.

235
Table 1.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Address</th>
<th>JMP location in RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI/IIC</td>
<td>$FFFF-5</td>
<td>$0209 (add: $20A/B)</td>
</tr>
<tr>
<td>Timer B</td>
<td>$FFFF-7</td>
<td>$0206 (add: $207/8)</td>
</tr>
<tr>
<td>Timer A</td>
<td>$FFFF-9</td>
<td>$0203 (add: $204/5)</td>
</tr>
<tr>
<td>IRQ</td>
<td>$FFFF-A</td>
<td>$0200 (add: $201/2)</td>
</tr>
<tr>
<td>SWI</td>
<td>$FFFC-D</td>
<td>used by monitor</td>
</tr>
<tr>
<td>RESET</td>
<td>$FFFF-F</td>
<td>use &quot;GO&quot; function</td>
</tr>
</tbody>
</table>

BREAKPOINTS

Up to three breakpoint addresses can be entered; this number can be easily increased if required by changing the EQU on line 25 of EBUG05 source code. To allow the continuation of execution from a breakpoint, a breakpoint at the start address specified by a GO is ignored. This means that a single breakpoint in a loop will only be encountered once if, after it has occurred, execution is re-commenced with a GO from the current address. If it is required to stop at the breakpoint again, then one of the two procedures described here should be used. Two breakpoints can be entered at different places in the loop. They will be encountered alternately; clearly, two GOs are required for each execution of the loop. A second method is to insert only one breakpoint but to trace one instruction before continuing with a GO. This trace takes the program counter past the breakpoint address and so allows the breakpoint to be re-inserted when GO is executed. This will only work reliably if the breakpoint is not on a branch, jump, or any other instruction that could cause the program to proceed to an address other than the next sequential address (see below).

If a single breakpoint has been entered and encountered and execution re-started with a GO from the same address, then pressing RESET is the only method of returning to the monitor. Normally pressing RESET when the program is running with breakpoints valid is not recommended as the application code will be left corrupted with SWIs replacing the instructions at the breakpoint addresses. If, however, control has been lost because execution was recommenced from the address of a single breakpoint, then corruption will not occur as the breakpoint will not have been re-inserted.

Breakpoints are inserted when TTrace or GO are executed but removed when a breakpoint is encountered. The trace facility inserts a breakpoint (SWI) at the address of the instruction sequentially after the current instruction and then executes the current instruction. If the current instruction jumps or branches somewhere else then the SWI will not be encountered. This is thus not a true program-following trace but in many cases is more useful as often subroutines do not need to be traced, variables being required to be checked only after each subroutine has been executed. If it is required to trace the program flow to the other address, then another breakpoint should be inserted at that address.
SERIAL LOAD

To load external Motorola S-records, the serial load key (L) should be pressed. The LCD will display "LOAD". S-records should then be supplied at 9600 baud (8-bit, no parity) on the RS232 interface. When an S9 termination record is received, the prompt returns. If an error is detected during a serial load, the load routine stops and displays the address at which the error occurred and the error type. The following error types are possible.

1: Checksum error, transmitted data or interface faulty.
2: RAM read-back error, RAM faulty or nonexistent.
3: ASCII character less than $30 (0) received.
4: ASCII character between $39 (9) and $41 (A) received.
5: ASCII character more than $46 (F) received.
7: Verify error when comparing S-records with RAM.

The verify function can be used to check that the RAM has not been corrupted. The VERIFY function is used exactly like LOAD except that RAM is compared with, rather than loaded by, the S-records.

The S-record format can be seen in the example below. The number following the S is the record type. The monitor only recognises S1 (8-bit data) and S9 (termination) records. The next byte (23 in most of the records) is the number (in hex) of bytes which follow. This is followed by a 2-byte address (e.g., E000 and FFF4), then the data. The last byte is a checksum byte. The sum of all bytes, including the byte count and the checksum byte, is $XXFF. This can be easily checked on the S9 record (03+00+00+FC=FF).

```
S123E000CDE05F24FB5F728D6E09FB128270BC1E0BF273A5C5C5C5C520EAA601B70E5CDC50
S123E020E09F6E3B7123F00A6F08705A658B701A6FBB7063F02A6FBB7073F03A61CB708FF
S123E5E0305A26F8AD3020EAA112705A11126E29981AE04BF2BB72D4444444497D6E4CD2C
S123E600BE28E720B62DA40F97D6E4CD6BBE28E721CDE1ECB62D81B72E9F2CB6305FADD5B61C
S10DE6202BEB2BDFB62EBEBEB146
S10FDF92E022B22E022E04EB022E022C5
S9030000FC
```

SERIAL INTERFACE

Figure 6 shows a suggested method of wiring up the RS232 sockets in a system with both loading and dumping capabilities. This arrangement facilitates use of the serial LOAD and DUMP routines of the monitor either via a PC COM port or between a host and terminal connected by an RS232 link. When using a PC, the “host” socket should be used. As only one pin on the MC68HC05E0 is used, switching is required to make the required connections. S2 can be eliminated (or left at “L”) if only loading is required, as will often be the case. To save power in battery applications, the RS232 interface chip can be switched off using S1. Table 2 shows possible methods of use.
Figure 6. RS232 serial interface with Load/Dump switching

Table 2.

<table>
<thead>
<tr>
<th>Set-up</th>
<th>Function</th>
<th>S1</th>
<th>S2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host &amp; terminal</td>
<td>Load</td>
<td>On</td>
<td>L</td>
<td>Terminal and host connected. Micro looks at data sent from host to terminal (pins 3).</td>
</tr>
<tr>
<td></td>
<td>Dump</td>
<td>On</td>
<td>D</td>
<td>Connection between terminal and host broken, S-records sent to both host (2) and terminal (3).</td>
</tr>
<tr>
<td>PC “COM” port</td>
<td>Load</td>
<td>On</td>
<td>L</td>
<td>S-records loaded from pin 3.</td>
</tr>
<tr>
<td></td>
<td>Dump</td>
<td>On</td>
<td>D</td>
<td>S-records sent to pin 2 on “host” socket (and pin 3 on “terminal” socket).</td>
</tr>
</tbody>
</table>

REFERENCES

1) AN441/D, An EPROM Emulator using the MC68HC05E0.

2) AN460/D, An RDS Decoder using the MC68HC05E0.
APPENDIX – EBUG05 E0 monitor listing

P. Topping 18th November '91

EBUG05 E0 monitor.

ORG $0020

PORTA EQU 0
PORTB EQU 1
PORTC EQU 2
PORTD EQU 3
PORTE EQU 4
PORTAD EQU 5
PORTBD EQU 6
PORTC EQU 7
PORTDD EQU 8
PORTED EQU 9
TCHR EQU $0C
PORTDSF EQU $12

ORG $0200

IRQ RMB 3
TIRQA RMB 3
TIRQB RMB 3
SIRQ RMB 3
DTABL RMB 6
WORK1 RMB 1
WORK4 RMB 1
TMP/CR RMB 1
PROP RMB 1

ORG $0E000

RMB 3*NBKPT B.P. TABLE
0063
    * Reset.
    *
    *

0064

0065

0066

0067

0068

0069

0070 e002 b7 05
    STA #FF
    JR KEYPAD

0071 e004 a6 ff
    LDA #.
    SETUP PORT

0072 e006 b7 12
    STA PORT F
    FOR ADDRESSES ETC.

0073

0074 e008 a6 cc
0075 e00a c7 02 00
0076 e00d c7 02 03
0077 e010 c7 02 06
0078 e013 c7 02 09
0079 e016 c6 e0 ac
0080 e019 c7 02 01
0081 e01c c6 e0 ad
0082 e01f c7 02 02
0083 e022 c6 e0 ae
0084 e025 c7 02 07
0085 e028 c6 e0 af
0086 e02b c7 02 08
0087 e02e c6 e0 b0
0088 e031 c7 02 04
0089 e034 c6 e0 b1
008a e037 c7 02 05
008b e03a c6 e0 b2
008c e03d c7 02 0a
008d e040 c6 e0 b3
008e e043 c7 02 0b
0090
e046 ae 20
0091
e049 7f
0092
e04c 23 fa
0093
e04e cd 5a
0094
e051 e6 1f
0095
e054 d7 02 16
0096
e057 5c
0097
e05a 4a 2e
0098
e05d 83
0099
e060 26 f6
0100
e063 3a 29
0101
e066 8f ff
0102
e069 5c
0103
e06c 5c
0104
e06f 5c
0105

0106
e072 5c
0107

0108
e075 5c
0109
e078 5c
0111

0112

240
0113          ******************************************************
0114          *                                                  *
0115          *          SWI.                                    *
0116          *                                                  *
0117          ******************************************************
0118          0119 e05e 00 20 04  SWI  BRSET,  0,STAT,SWICHK FROM RESET?
0119 e061 10 20  BSET  0,STAT  YES
0120 e063 20 4f  BRA  GETCMD
0121 e065 b6 0c  SWICHK  LDA  TCR  GET CURRENT TCR
0122 e067 ae 0c  LDX  #$20C  SET XROM SO THAT EXTERNAL RAM
0123 e069 bf 0c  STX  TCR  CAN BE WRITTEN TO
0124 e06b c7 02 14  STA  TMPTCR  SAVE CURRENT TCR
0125 e06c cd e6 74  STAY  JSR  KEYSRN
0126 e071 cd e6 1f  JSR  SCNRBP  REMOVE
0127 e074 d6 02 16  SWIREP  LDA  BKPTBL,X  BREAKPOINTS
0128 e077 2b 0d  BMII  SWINOB
0129 e079 b7 22  STA  ADOIIH
0130 e07b d6 02 17  LDA  BKPTBL+1,X
0131 e07e b7 23  STA  ADOIRL
0132 e080 d6 02 18  LDA  BKPTBL+2,X
0133 e083 cd e7 57  JSR  STORE
0134 e086 5c  SWINOB  INCX  GET NEXT B.P.
0135 e087 5c  INCX
0136 e088 5c  INCX
0137 e089 3a 29  DEC  PNINT
0138 e08b 26 e7  BNE  SWIREP
0139 e08d cd e1 23  JSR  LOCSTK  FIND STACK
0140 e090 e6 08  LDA  8,X
0141 e092 a0 01  SUB  #1  ADJUST PC
0142 e094 e7 08  STA  8,X  (MINUS 1)
0143 e096 b7 23  STA  ADOIRL
0144 e098 e6 07  LDA  7,X
0145 e09a 2a 00  SBC  #0
0146 e09c e7 07  STA  7,X
0147 e09e b7 22  STA  ADOIIH
0148 e0a0 03 20 06  BRCLR  1,STAT,NOPRO  PROCEED?
0149 e0a3 c6 02 15  LDA  PROP
0150 e0a6 cd e7 57  JSR  STORE
0151 e0a9 cc e1 35  NOPRO  JMP  PCOUNT  PRINT P.C.
0152 e0ac e8 9c  VECTOR  FDB  IRQV  IRQ
0153 e0ae e8 9c  FDB  TIRQBV  TIMER B
0154 e0b0 e8 9c  FDB  TIRQAV  TIMER A
0155 e0b2  e8 9c  FDB  SIRQV  SERIAL

241
0257 e135 a6 73
0258 e137 c7 02 10
0259 e13a a6 d1
0260 e13c c7 02 11
0261 e13f ad e2
0262 e141 e6 07
0263 e143 b7 22
0264 e145 e6 08
0265 e147 b7 23
0266 e149 cd e8 02
0267 e14c cc e0 bf
0268 e151 c7 02 0c
0269 e154 c7 02 0f
0270 e157 a6 d1
0271 e159 c7 02 0d
0272 e15c c7 02 0e
0273 e15f ad e2
0274 e161 9f
0275 e162 ab 05
0276 e164 3f 22
0277 e166 b7 23
0278 e168 lc 20
0279 e16a cc e6 ea
027a e170 a6 06
027b e172 c7 02 0d
027c e175 a6 e6
027d e177 c7 02 0e
027e e17a a6 60
027f e17c c7 02 0f
0280 e17f ad e2
0281 e181 9f
0282 e182 ab 06
0283 e184 3f 22
0284 e186 b7 23
0285 e188 lc 20
0286 e18a cc e6 ea
0287 e190 a6 dl
0288 e192 c7 02 0c
0289 e194 9f
028a e195 b7 23
028b e197 c7 02 0d
028c e19c c7 02 0e
028d e19f a6 f1
028e e1a1 c7 02 0f
028f e1a4 cc e6 ea
0290 e1a7 9f
0291 e1a8 ab 04
0292 e1aa 3f 22
0293 e1ac b7 23
0294 e1ae cc e0 cc
0295 e1af cc e0 cc
0296 e1b0 cc e6 ea
0297 e1b1 cc e0 be
0298 e1b2 cc e0 be
0299 e1b3 cc e0 be
029a e1b4 cc e0 be
029b e1b5 cc e0 be
029c e1b6 cc e0 be
029d e1b7 cc e0 be
029e e1b8 cc e0 be
029f e1b9 cc e0 be
0300 e1ba cc e0 be
0301 e1bb cc e0 be
0302 e1bc cc e0 be
0303 e1bd cc e0 be
0304 e1be cc e0 be
0305 e1bf cc e0 be
0306 e1c0 cc e0 be
0307 e1c1 cc e0 be
0308 e1c2 cc e0 be
0309 e1c3 cc e0 be
030a e1c4 cc e0 be
030b e1c5 cc e0 be
030c e1c6 cc e0 be
030d e1c7 cc e0 be
030e e1c8 cc e0 be
030f e1c9 cc e0 be
0310 e1ca cc e0 be
0311 e1cb cc e0 be
0312 e1cc cc e0 be
0313 e1cd cc e0 be
0314 e1ce cc e0 be
0315 e1cf cc e0 be
0316 e1d0 cc e0 be
0317 e1d1 cc e0 be
0318 e1d2 cc e0 be
0319 e1d3 cc e0 be
031a e1d4 cc e0 be
031b e1d5 cc e0 be
031c e1d6 cc e0 be
031d e1d7 cc e0 be
031e e1d8 cc e0 be
031f e1d9 cc e0 be
Build a beginning and ending address range.

TEMP, TEMP+1 - ADDRH, ADDRL.

BLDRN: BCLR 4, STAT
JSR CLRTAB PRINT
LDA #$F4 "BA"
STA DTABLE+4
LDA #$77
STA DTABLE+5
JSR DISTAB
JSR BLADDR GET SOURCE ADDR.
BCC BLDRN1 VALID?
LDA ADDRH YES
STA TEMP NO SAVE IT

BLDRN1: BSET 4, STAT INVALID
RTS

DISP STX WORK6
CLR COUNT
DISLP LDX WORK6
INC WORK6
INC COUNT
LDA COUNT
LDA DLOAD, X
LDA DLOAD, X
STA DTABLE, X
INC WORK6
INC COUNT
LDA COUNT
LDA DLOAD, X
CMP #6
JMP DISTAB
BLC DISLP
JMP DISTAB

DLOAD FCB 0,0,$D0,$D7,$77,$7B
VERF FCB $D6,$F1,$60,$60,$61,$6B

244
Calculate branch offset.
0447 If **
0448 Branch offset (continued).
0449 *
0450
0451 STA DTABL+1
0452 LDA #$F1
0453 STA DTABL+2
0454 LDA #$FE
0455 STA DTABL+3
0456 LDA ADDR
0457 JSR PRINTOFFSET
0458 TXX
0459 LDA TEMP+1
0460 ADD #$1
0461 STA ADDR
0462 LDA TEMP
0463 ADC #0
0464 STA ADDR
0465 PUT INTO
0466 STA ADDR
0467 INSTRUCTION
0468 JMP STORE
0469
0470 OFFST1 LDA ADDR
0471 OFFST1 LDA ADDR
0472 STA ADDR
0473 LDA ADDR
0474 STA ADDR
0475 LDA ADDR
0476 CMP #$FF
0477 OFFST1 BEQ OFFST3
0478 TSTA CHECK FOR
0479 OFFST2 BNE OVRERR +/ - 0 AND -1
0480 OFFST3 LDA ADDR
0481 CMP #$7F
0482 BHI OVRERR
0483 BRA OK2
0484 OFFST3 LDA ADDR
0485 OFFST3 LDA ADDR
0486 STA ADDR
0487 CMP #$FF
0488 BEQ OVRERR
0489 CMP #$FE
0490 BEQ OVRERR
0491 CMP #$FF
0492 BHI OVRERR
0493 OK2 INC TEMP+1
0494 OFFST3 BNE OFFST
0495 OFFST3 INC TEMP
0496 OFFST3 OFFST USE PRINT IF VALID
0497 BRA CMDJMP
0498 OVRERR LDA #$07
0499 PRINT OR
0500 STA DTABL+4
0501 LDA #$60
0502 STA DTABL+5
0503 JSR PRINTDE
0504 CMDJMP JMP CMDSCN
0505
0506
0507
0508
0509
0510
0511
0512
0513 e2cf 3f 26 BPDIS CLR WORK6
0514 e2d1 3a 26 DEC WORK6
0515 e2d3 cd e6 1f JSR SCNBKP
0516 e2d6 bf 21 STX WORK2
0517 e2d8 4f BPDIS1 CLAA
0518 e2e1 a6 f4 STX >ORK6
0519 e2e3 c7 02 10 STA DTABL+4
0520 e2e5 c7 02 14 LDA BKPTBL,X GET B.P.
0521 e2e7 c7 02 0c STA DTABL PRINT 'BOFF'
0522 e2e9 c7 02 0d STA DTABL+1
0523 e2eb a6 71 LDA #$71
0524 e2ed c7 02 0e STA DTABL+2
0525 e2f3 2a 14 STA DTABL+3
0526 e2f5 b7 22 BPDIS2 STA ADDR PRINT B.P.
0527 e2f7 27 19 BRA BPOIS YES
0528 e2f9 be 21 STX >ORK2
0529 e301 a6 f1 LOA CTABL.X NEW B.P.
0530 e303 d6 e7 6e LDA CTABL.X STORE NEW B.P.
0531 e305 c7 02 11 STA DTABL+5
0532 e307 e2f0 3c 26 BPDIS4 INC WORK6 PRINT B.P.
0533 e309 be 26 LDX WORK6
0534 e30b 3a 29 STX WORK2
0535 e30d bf 21 STX WORK2
0536 e30f cc e4 42 BPRET JMP GETCMD
0537 e313 a6 ff JSR CLRTAB
0538 e315 a6 60 STA DTABL+1
0539 e317 c7 02 0d STA DTABL+2
0540 e319 27 0a BPDIS7 LDA ADDR
0541 e31b 27 19 STA BKPTBL,X
0542 e31d 27 19 STA BKPTBL+1,X
0543 e31f 27 19 STA BKPTBL+2,X
0544 e321 af 26 BRA BPDIS
0545 e323 af 26 INCX
0546 e325 5c INCX
0547 e326 b6 22 BEQ BPOIS5
0548 e328 c7 02 17 BPDIS5 LDA ADDR
0549 e32a bd 22 STA BKPTBL,X
0550 e32c be 22 STA BKPTBL+1,X
0551 e32e be 22 STA BKPTBL++1,X
0552 e330 cc e0 b4 BPRET JMP GETCMD
0553 e332 cd e6 42 ERROR JSR CLRTAB
0554 e334 cd e6 42
0555 e336 2a 14
0556 e338 c7 02 0d
0557 e33a c7 02 0e
0558 e33c 00 0f
0559 e33e 00 0d
0560 e336 00 0c
0561 e33e 00 0b
0562 e33f 00 0a
0563 e340 00 09
0564 e341 00 08
0565 e342 00 07
0566 e343 00 06
0567 e344 00 05
0568 e345 00 04
0569 e346 00 03
0570 e347 00 02
0571 e348 00 01
0572 e349 00 00
247
Breakpoint clear.
RS232 (9600) S-Record receiver (EO at 4MHz).
Input routine, MC68HC05E0 : 0.5 uS.

Cycles per bit at 9600 baud : 208.

** ***************************************

** INCHD BSR DEL191 191 GET OUT OF BIT 7
** INCH BCCLR 2, PORTB,* 5 IS LINE HIGH ?
** BRSET 2, PORTB,* 5 YES, WAIT FOR START
** LDX #7 2 6 7 DATA BITS TO READ
** STX COUNT 4 10 WAIT TILL MIDDLE OF 1ST BIT
** BSR DEL110 110 120 */-3

** INBT BSR DEL191 191 */+120-208=103
** BCCLR 2, PORTB, ZER 5 196 CYC 2 (105) READ
** ZER RORA 3 199 SAVE BIT
** DEC COUNT 5 204
** BNE INBT 3 207

** LSRA 3 16 MSB A ZERO
** RTS 6 22

** NINE JMP GETCMD
** ERR2 LDX #2 RAM READBACK
** BRA ERR
** ERR3 LDX #3 LESS THAN ASCII 0
** BRA ERR
** ERR4 LDX #4 BETWEEN ASCII 9 & A
** BRA ERR
** ERR5 LDX #5 MORE THAN ASCII F
** BRA ERR
** ERR7 LDX #7 VERIFY ERROR
** BRA ERR

** 251
### Byte input sub-routines

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Byte</th>
<th>Nibble</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR</td>
<td>INCHD</td>
<td>22</td>
<td>MS NIBBLE</td>
</tr>
<tr>
<td>BSR</td>
<td>ASCII</td>
<td>35</td>
<td>WHAT WAS IT</td>
</tr>
<tr>
<td>LSLA</td>
<td></td>
<td>3</td>
<td>OK</td>
</tr>
<tr>
<td>LSLA</td>
<td></td>
<td>3</td>
<td>SHIFT</td>
</tr>
<tr>
<td>LSLA</td>
<td></td>
<td>3</td>
<td>IT</td>
</tr>
<tr>
<td>STA</td>
<td>TMP1</td>
<td>4</td>
<td>73</td>
</tr>
<tr>
<td>LDA</td>
<td>TMP2</td>
<td>3</td>
<td>76</td>
</tr>
<tr>
<td>ADD</td>
<td>CHKSUM</td>
<td>3</td>
<td>79</td>
</tr>
<tr>
<td>STA</td>
<td>CHKSUM</td>
<td>4</td>
<td>83</td>
</tr>
<tr>
<td>BSR</td>
<td>INCHD</td>
<td>22</td>
<td>LS NIBBLE</td>
</tr>
<tr>
<td>BSR</td>
<td>ASCII</td>
<td>35</td>
<td>57</td>
</tr>
<tr>
<td>ADD</td>
<td>TMP1</td>
<td>3</td>
<td>60</td>
</tr>
<tr>
<td>STA</td>
<td>TMP2</td>
<td>4</td>
<td>64</td>
</tr>
<tr>
<td>DEC</td>
<td>BCONT</td>
<td>5</td>
<td>69</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td>6</td>
<td>75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Byte</th>
<th>Nibble</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>CMP</td>
<td>2</td>
<td>BEFORE ZERO</td>
</tr>
<tr>
<td>ADD</td>
<td>ERR3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>CMP</td>
<td>$39</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>BHI</td>
<td>MTS</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>SUB</td>
<td>$30</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td>6</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Byte</th>
<th>Nibble</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTH</td>
<td>CMP</td>
<td>2</td>
<td>BEFORE A</td>
</tr>
<tr>
<td>ADD</td>
<td>ERR4</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>CMP</td>
<td>$46</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>BHI</td>
<td>ERR5</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>SUB</td>
<td>$17</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td>6</td>
<td>29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Byte</th>
<th>Nibble</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL191</td>
<td>LDX</td>
<td>2</td>
<td>#29</td>
</tr>
<tr>
<td>BRA</td>
<td>DELAY</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>DEL110</td>
<td>LDX</td>
<td>2</td>
<td>#16</td>
</tr>
<tr>
<td>DELAY</td>
<td>DECX</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>DELAY</td>
<td>BNE</td>
<td>3</td>
<td>5 X X</td>
</tr>
<tr>
<td>PTS</td>
<td></td>
<td>6</td>
<td>12+AX (INC BSR)</td>
</tr>
</tbody>
</table>

---

252
0810 RS232 (9600 @ 4MHz) S-Record output.

0820

0821

0822

0823

0824

0825

0826

0827

0828

0829

0830

0831

0832

0833

0834

0835

0836

0837

0838

0839

0840

0841

0842

0843

0844

0845

0846

0847

0848

0849

0850

0851

0852

0853

0854

0855

0856

0857

0858

0859

0860

0861

0862

0863

0864

0865

0866

0867

0868

0869

0870

0871

0872

0873

0874

* Checksum byte.

*

* Finished?
0875
0876
0877
0878
0879
0880
0881 e52c a6 53
0882 e52e ad 33
0883 e530 a6 39
0884 e532 ad 2f
0885 e534 a6 03
0886 e536 ad 57
0887 e538 a6 00
0888 e53a ad 53
0889 e53c a6 00
0890 e53e ad 4f
0891 e540 a6 fc
0892 e542 ad 4b
0893 e544 ad 17
0894 e546 15 06
0895 e548 cc e0 b4
0896
0897
0898
0899
0900
0901
0902
0903 e54b 0a 20 09
0904 e54e ae c7
0905 e550 bf 21
0906 e552 bd 21
0907 e554 c7 02 13
0908 e557 ae c6
0909 e559 bf 21
0910 e55b bc 21
0911
0912
0913 e55d a6 0d
0914 e55f ad 02
0915 e561 a6 0a
0916
0917
0918
0919
0920
0921
0922
0923
0924 e563 14 01
0925 e565 ae 0a
0926 e567 bf 2b
0927 e56e cd e4 c1
0928 e56e 98
0929 e56d 20 08
0930
0931 e56f ae la
0932 e571 cd e4 c1
0933 e574 9d
0934 e575 9b
0935 e576 46
0936 e577 25 04
0937 e579 15 01
0938 e57b 20 04
0939 e57d 14 01
0940 e57f 20 00
0941 e581 3a 2b
0942 e583 26 ea
0943 e585 81
0944
0945 e586 ab 30
0946 e588 a1 39
0947 e58a 23 02
0948 e58c ab 07
0949 e58e 81
0950

--------------------------------------------------------------------------
  * S9 record. *
--------------------------------------------------------------------------

LDA # .S' S
BSR OUCH
LDA # .9' 9
BSR OUCH
LDA #$03 3 BYTES
BSR BYTED
LDA #$00
BSR BYTED DUMMY (0)
BSR BYTED ADDRESS
LDA #$FC
BSR BYTED CHECKSUM
BSR CRLF
BCLR 2, PORTB BIT 2 INPUT
JMP GETCMD

--------------------------------------------------------------------------
  * RAM R/W for S-record load. *
--------------------------------------------------------------------------

RAMACC BSET 5, STAT, L3 5 READING ?
LDX #$C7
STX WORK2
JSR WORK2
STA WORK4
L3
LDX #$C6
STX WORK2
JMP WORK2 13 50 (56 WITH JSR)

--------------------------------------------------------------------------
  * Output routine, 208 cycles per bit. *
--------------------------------------------------------------------------

OCHL RSET 2, PORTB 5 MAKE SURE IT'S HIGH
LDX #10
STX COUNT
JSR DELAY 72 83
CLC
BRA STAR 3 88

--------------------------------------------------------------------------
  * DEL3 puts on ones for STOP *
--------------------------------------------------------------------------

OUTBT LDX #28
JSR DELAY 180 182
NOP
DEL3 SEC
RORA 3 189 GET A BIT
STAR BCS QUIT 3 192 YES
BCLR 2, PORTB 5 197 NO
BRA OBD 3 200
OBD DEC COUNT
BNE OUTBT 3 208 DONE ?
RTS

--------------------------------------------------------------------------
  * ASCII are written this way *
--------------------------------------------------------------------------

ASCI0 ADD #$30
CMP #$39
BLS NMT9
ADD #$07
RTS

--------------------------------------------------------------------------
  * RAM core, 208 cycles per bit *
--------------------------------------------------------------------------

254
0951 0952 0953 0954 0955 0956
0957 e58f b7 2c
0958 e591 44
0959 e592 44
0960 e593 44
0961 e594 44
0962 e595 ad ef
0963 e597 ad ca
0964 e599 b6 2c
0965 e59b b2 2a
0966 e59d b7 2a
0967 e59f b6 2c
0968 e6a1 a4 0f
0969 e6a3 ad el
0970 e6a5 ad bc
0971 e6a7 3a 2e
0972 e6a9 81
0973 0974

******************************************************************************
* Byte output sub-routine. *
******************************************************************************
BYTEO STA TMP1
LSRA SHIFT
LSRA DOWN TO
LSRA GET MSB
LSRA
BSR ASCIO & CONVERT
BSR OUCH
LDA TMP1 RESTORE BYTE
ADD CHKSUM ACCUMULATE
STA CHKSUM IN CHECKSUM BYTE
LDA TMP1
AND #$0F LSB
BSR ASCIO CONVERT IT
BSR OUCH
DEC BCNT DECREMENT BYTE COUNT
RTS

255
NEWGO BCLR 1,STAT

JSR LOCSTK

LDA 8,X

STA ADDRRL

LDA 7,X

STA ADDRPH

JSR GETADR

ADDRESS VALID?

BCS GOON

CMP #310

BEQ GOBACK

CMP #511

BNE GONERR

JSR WCSTK

YES

BNE SWI

IF NOT THEN INSERT SWI

LDA BKPTBL+1,X SAME,

GET LSB

CMP TEMP+1 SAME AS THAT OF NEXT ADDRESS?

BEQ GONOB2

IF SO THEN SWI

LOA BKPTBL1-1,X SAME,

GET LSB

CMP TEMP SAME AS MSB OF NEXT ADDRESS?

BNE GONOB2

IF NOT THEN INSERT SWI

LDA #58

JSR STORE

PNCNT

BNE GONOB2 DONE?

GOOB2 INCX

GET NEXT B.P.

INCX

INCX

DECC

PNCNT

BNE GOINSB DONE?

GOINSB LDA BKPTBL.X GET ADDRESS MSB

BMI GONOB VALID? (<32K ONLY)

STA ADDRPH YES

LDA BKPTBL+1.X LSB

STA ADDRRL

JSR LOAD READ INSTRUCTION

STA BKPTBL+2,X

GONOa INCX GET NEXT 8.P.

INCX

INCX

DEC PNCNT

RESTART

BMI GONOB

SAME AS MSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS MSB OF NEXT ADDRESS?

SAME. GET LSB

IF NOT THEN INSERT SWI

IF SO THEN NO SWI

LDA BKPTBL+1.X SAME, GET LSB

STA GONOB2 IF SO THEN NO SWI

STA ADDRRL

LDA #58

JSR STORE

PNCNT

BNE GOINSB DONE?

GOINSB LDA BKPTBL.X INSERT B.P.'S

BMI GONOB YES

LDA BKPTBL+1.X LSB

STA ADDRRL

JSR LOA READ INSTRUCTION

STA ADDRPH

LDA BKPTBL+2,X

GONOa INCX GET NEXT 8.P.

INCX

INCX

DEC PNCNT

RESTART

BMI GONOB

SAME AS MSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS MSB OF NEXT ADDRESS?

SAME. GET LSB

IF NOT THEN INSERT SWI

IF SO THEN NO SWI

LDA BKPTBL+1.X SAME, GET LSB

STA GONOB2 IF SO THEN NO SWI

STA ADDRRL

LDA #58

JSR STORE

PNCNT

BNE GOINSB DONE?

GOINSB LDA BKPTBL.X INSERT B.P.'S

BMI GONOB YES

LDA BKPTBL+1.X LSB

STA ADDRRL

JSR LOA READ INSTRUCTION

STA ADDRPH

LDA BKPTBL+2,X

GONOa INCX GET NEXT 8.P.

INCX

INCX

DEC PNCNT

RESTART

BMI GONOB

SAME AS MSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS MSB OF NEXT ADDRESS?

SAME. GET LSB

IF NOT THEN INSERT SWI

IF SO THEN NO SWI

LDA BKPTBL+1.X SAME, GET LSB

STA GONOB2 IF SO THEN NO SWI

STA ADDRRL

LDA #58

JSR STORE

PNCNT

BNE GOINSB DONE?

GOINSB LDA BKPTBL.X INSERT B.P.'S

BMI GONOB YES

LDA BKPTBL+1.X LSB

STA ADDRRL

JSR LOA READ INSTRUCTION

STA ADDRPH

LDA BKPTBL+2,X

GONOa INCX GET NEXT 8.P.

INCX

INCX

DEC PNCNT

RESTART

BMI GONOB

SAME AS MSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS MSB OF NEXT ADDRESS?

SAME. GET LSB

IF NOT THEN INSERT SWI

IF SO THEN NO SWI

LDA BKPTBL+1.X SAME, GET LSB

STA GONOB2 IF SO THEN NO SWI

STA ADDRRL

LDA #58

JSR STORE

PNCNT

BNE GOINSB DONE?

GOINSB LDA BKPTBL.X INSERT B.P.'S

BMI GONOB YES

LDA BKPTBL+1.X LSB

STA ADDRRL

JSR LOA READ INSTRUCTION

STA ADDRPH

LDA BKPTBL+2,X

GONOa INCX GET NEXT 8.P.

INCX

INCX

DEC PNCNT

RESTART

BMI GONOB

SAME AS MSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?

SAME AS MSB OF NEXT ADDRESS?

SAME. GET LSB

IF NOT THEN INSERT SWI

IF SO THEN NO SWI

LDA BKPTBL+1.X SAME, GET LSB

STA GONOB2 IF SO THEN NO SWI

STA ADDRRL

LDA #58

JSR STORE

PNCNT

BNE GOINSB DONE?

GOINSB LDA BKPTBL.X INSERT B.P.'S

BMI GONOB YES

LDA BKPTBL+1.X LSB

STA ADDRRL

JSR LOA READ INSTRUCTION

STA ADDRPH

LDA BKPTBL+2,X

GONOa INCX GET NEXT 8.P.

INCX

INCX

DEC PNCNT

RESTART

BMI GONOB

SAME AS MSB OF NEXT ADDRESS?

SAME AS LSB OF NEXT ADDRESS?
**Keypad scan, carry set if valid output.**

KEYSCN  CLC  
CLRA  
LDX  $6  
SETUP  
KEY1  ADD  #$10  
ROW  
STA  PORTA  
CHECK COLUMN  
BSC  KEY2  IF VALID GET OUT  
DECX  ELSE TRY  
BNE  KEY1  NEXT ROW  
KEY2  RTS  

**Check for key closure.**

COLUMN  LDA  PORTA  READ KEYPAD  
STA  WORK1  STORE IT  
BIT  #$0F  KEY CLOSED?  
BEQ  COLRET  NO GET OUT  
BSC  DBOUNC  ELSE DEBOUNCE  
LOA  PORTA  RE-READ KEYPAD  
CMP  WORK1  SAME KEY CLOSED?  
BNE  COLRET  NO GET OUT  
SEC  SET FLAG FOR VALID  
COL1  LDA  PORTA  KEY  
BIT  #$0F  RELEASED?  
BNE  COL1  NO TRY AGAIN  
BSC  DBOUNC  YES DEBOUNCE  
LOA  PORTA  STILL  
BIT  #$0F  RELEASED?  
BNE  COL1  NO TRY AGAIN  
LOA  WORK1  RETURN CHAR IN A-REG  
COLRET  RTS  YES GO HOME  

**Pause for 3075 cycles.**

DBOUNC  LDA  $10  40MS  
STA  WORK5  
DLP  LDA  OFFSET  PAUSE  
DLOOP  BRN  *  256X12  
BRN  *  CYCLES  
DECA  OR AT  
DEC  WORK5  
BNE  DLP  LEAST 3.7ms  
BNE  DLP  
PTS  

**Input one character, A contains value.**

CHRIN  JSR  KEYSCN  GET KEY  
BCC  CHRIN  IF NOT VALID RETRY  
CHRIN1  CMP  STABILX  CONVERT  
BEQ  CHRIN2  TO HEX  
INX  
BRA  CHRI  
CHRIN2  TAX  IF CANCEL  
RTS  

258
Load byte at ADDRH,ADDRL into A.

```
LOAD STX WORK1 SETUP
LDX #$C6 ROUTINE
LDSTCM STX WORK2 TO DO
LDX #$81 TWO BYTE
STX WORK3 LOAD
JSR WORK2
LDX WORK1
RTS
```

Store byte in A at ADDRH,ADDRL.

```
STORE STX WORK1
LDX #$C7 SETUP
BSR LDSTCM ROUTINE
STA WORK4 DO
JSR LOAD TWO BYTE
CMP WORK4 STORE
BEC STRTS
SEC
STA WORK1
RTS
```

Hex. to mux. display conversion.

```
CTABL FCB $D7 0
FCB $96 1
FCB $E3 2
FCB $A7 3
FCB $36 4
FCB $35 5
FCB $F5 6
FCB $97 7
FCB $F7 8
FCB $87 9
FCB $77 A
FCB $F4 B
FCB $D1 C
FCB $E6 D
FCB $F1 E
FCB $71 F
```

Build a byte in accumulator.

```
GETBY2 STA WORK2
BSR GETMYB
BCC GETBRT
ASL WORK2
ASL WORK2
ASL WORK2
ORA WORK2
SEC
GETBRT RTS
```
1358
1359
1360
1361
1362
1363
1364
1365
1366 e790 cd e6 bb
1367 e791 98
1368 e794 a1 0f
1369 e796 22 01
1370 e798 99
1371 e799 81

**... ...
...** ...
.. ...
...

1372
1373
1374
1375
1376
1377
1378
1379
1380
1381 e79a cd e6 42
1382 e79d cd e8 02
1383 e7a4 ad ee
1384 e7a7 ae 04
1385 e7a8 af 02
1386 e7a9 ad ee
1387 e7aa af 02
1388 e7ab ae 04
1389 e7ac af 02

1390 e7a4 af 02
1391 e7a6 ae 04
1392 e7a7 af 02
1393 e7a8 af 02
1394 e7a9 ad ee
1395 e7aa af 02
1396 e7ab ae 04
1397 e7ac af 02
Transfer code from EPROM to RAM, default destination address: 0400-$1FFF, from EPROM at $4400 - $5FFF.

(TEMP, TEMP+1 -> ADDRH, ADDRL).

Print address ADDRH, ADDRL.

Transfer code from EPROM to RAM, default destination address: 0400-$1FFF, from EPROM at $4400 - $5FFF.

(TEMP, TEMP+1 -> ADDRH, ADDRL).

Print address ADDRH, ADDRL.
Transfer code from EPROM to RAM.

XLOOP
LDA ADDRH
ADD $40
STA ADDRH
JSR LOAD
TAX
LDA ADDRH
SUB $40
STA ADDRH
TXA
JSR STORE
BCC RXOK
JMP ERR2
RXOK INC ADDRL
INC ADDRH
XSKP LDA ADDRH
CMP TEMP
BNE XLOOP
CMP TEMP+1
BNE XLOOP
XAB JMP GETCMD

Vectors etc.

RTI
JSR CLRDIS
STOP
BRA STP
JSR CLRDIS
WAIT
BRA WIT
ORG $FFF4
FDB RESET SERIAL
FDB TIQB TIMER B
FDB TIQCA TIMER A
FDB IRQ EXTERNAL INTERRUPT
FDB SWI SWI
FDB RESET RESET
END
INTRODUCTION

The low voltage inhibit (LVI) option, as used with many of the M6805 HMOS Family Microcomputer (MCU) devices (EPROM MCUs excluded), provides a means for the MCU to sense a drop in supply voltage (VCC) and then shut itself down in a well-defined manner. The LVI option may be used in applications which require the correct-output during normal operation and no-output during loss of power. The LVI option is also useful in fail-safe and/or fall-back operating modes for minimum systems operation.

An example of this no-output control scheme is in height-positioning applications. With correct-output control signals from the MCU, the position of "cherry-picker" platforms could be controlled and maintained. With inadvertant loss of power due to power supply/battery deterioration and/or cable disconnects, the no-output signal would permit locking methods to be initiated within microseconds. This would allow the "cherry-picker" to maintain its "last position" (thus preventing disastrous falls for personnel or on-board equipment) or to start a predetermined (hydraulic self-locking or hydraulic bleed-off valves) controlled descent.

This LVI capability of transition from correct-output to no-output (Le., high-Z, 3-state) without an intermediate uncontrollable region is defined in the On-Chip Operation paragraph.

Another example of this correct-output usefulness is in starter-controls of multi-horsepower electric motors. During normal power transitions, the controller signals will start up the motor at prescribed voltage and current versus time relationships, and maintain specified shaft RPM and shaft output power afterwards. At low power conditions (i.e., brownout), the electric current required to maintain this same motor-rpm and shaft output power increases proportional to the voltage decrease; e.g., a 20% drop in powerline voltage causes an equivalent 20% increase in motor current. This 20% increase in motor current produces a 44% increase in internal $I^2R$ loss. The internal $I^2R$ loss may cause the motor to overheat or even burn out.

The M6805 HMOS MCU can detect this "brownout" (low voltage) condition and immediately place the MCU output control lines (normally used to turn on the motor controller) in the high-impedance state. Bipolar driver devices with built-in pull-down resistors are usually used in these applications to turn themselves off when the control input is open-circuited.

Restart can be then prohibited until normal power conditions return and the original current versus time relationships are reestablished, within the controller.

The LVI option is provided at the time of manufacture by on-chip circuitry, as a mask option, contained in part of the users ROM pattern. When the LVI option is provided, no additional external parts are required for normal operation.

The LVI option will usually provide for an overall product cost reduction by eliminating the external components required to implement this feature off-chip.

ON-CHIP RESET/LVI OPERATION

A simplified equivalent of the internal reset and LVI circuitry is shown in Figure 1. The circuit consists of three basic sections: (1) internal reset generator, (2) an internal Schmitt trigger which is externally activated, plus bias circuits, clamping diodes and current limiting, and (3) a low voltage detector with gating logic.

LVI DISABLED

Without the LVI option (i.e., LVI disabled), the internal reset generator is only actuated by the external RESET pin via the Schmitt trigger. In this case during power-on reset (POR), external capacitor CR (0.1 to 1.0 microfarad) is charged through an on-chip resistor (Rc) and the current source from VCC. When the RESET pin voltage rises to the Schmitt trigger positive threshold ($V_{\text{RES}+}$), the on-chip reset generator allows the CPU to begin executing from ROM. The RC delay ($t_{\text{RH}}$) allows the on-chip oscillator to stabilize prior to start of program execution. During normal power turn-off, the on-chip reset generator is actuated by the
Schmitt trigger after the decreasing voltage on the \texttt{RESET} pin falls to the negative threshold voltage (\texttt{V\textsubscript{IRES} \textsubscript{--}}). The individual M6805 HMOS Family data sheet and Figure 2 (of this application note) contain information concerning the reset and LVI timing waveforms, and \texttt{V\textsubscript{CC}} voltage spectrum.

\textbf{LVI ENABLED}

With the LVI mask option (i.e., LVI enabled), the power-on sequence is exactly the same as described above. However, in the power-down condition (resulting from normal power turn off, brownout, or voltage "dip"), the on-chip reset generator is triggered by the low voltage detector before power falls below the reset level. In this case (as shown in Figure 1), the second input to the reset generator OR gate becomes functional and the low voltage detect circuit causes a reset at a voltage point (\texttt{V\textsubscript{LVI}}) prior to the \texttt{RESET} pin reaching the Schmitt trigger \texttt{V\textsubscript{IRES} \textsubscript{--}} threshold. The only requirement is that \texttt{V\textsubscript{LVI}} remains at its threshold for one \texttt{t\textsubscript{cyc}} (minimum). In typical applications, the \texttt{V\textsubscript{CC}} bus filter capacitor will eliminate negative-going voltage glitches of less than one \texttt{t\textsubscript{cyc}}. Once the \texttt{V\textsubscript{LVI}} threshold is reached for one \texttt{t\textsubscript{cyc}}, the low voltage detect circuit outputs a logic 1 which is gated to the on-chip reset generator, resetting the CPU within the next \texttt{t\textsubscript{cyc}} period.

Simultaneous to the activation of the on-chip reset generator, the low voltage detector turns on the internal \texttt{R\textsubscript{ON}} device. With the \texttt{R\textsubscript{ON}} device turned on, the external reset capacitor discharges through internal current limiter \texttt{RD} and continues until the \texttt{RESET} pin voltage falls below the minimum reset voltage, holding the CPU in reset. This condition remains until recovery of \texttt{V\textsubscript{CC}}, at which time normal power-on reset resumes.
LVl TESTING

Figure 3 shows an LVl test circuit connection for the MC680SP4L1 (or P1). Similar connections for other M6805 HMOS Family MCUs could be made to corresponding pins for LVl testing. This circuit, together with the software in Figure 4, is used to determine the $V_{LVI}$ and $V_{LVR}$ for the MCU under test.

Figure 4 is the test pattern (TEST P) software routine to be entered into RAM in order to generate a continuous output square wave on the PBO pin. By utilizing the on-chip monitor capability of the MC680SP4L1 demonstration program, the TEST P program can be loaded into on-chip RAM via an RS-232-C terminal. This connection is shown schematically in Figure 3. In typical lab applications, the 15 k pullup resistor is adequate to provide the required operating frequency. Alternatively, an adjustable resistor may be used to set the frequency at 3.58 MHz.

In order to activate the LVl state, the supply voltage ($V_{CC}$) must drop below $V_{LVI}$ and remain there for one $t_{VCC}$ (internal clock period) plus 250 nanoseconds. The M6805 HMOS Family $V_{CC}$ voltage spectrum is shown in Figure 2b. To determine the $V_{LVI}$ trip point, reduce $V_{CC}$ from the normal value (5.25 Vdc) to the point where the TEST P square wave disappears (i.e., PBO switches to high impedance). This is the $V_{LVI}$ value for this particular device. When $V_{LVI}$ is attained by reducing $V_{CC}$, the three-state leakage current ($I_{TS}$) can then be measured to check that all I/O port pins are in the high-impedance state.

Increase $V_{CC}$ in small increments (10-100 millivolt steps) and reinitiate the test pattern TEST P by keying the Execute address $040$ into the RS-232-C terminal. This determines the $V_{LVR}$ value for the device being tested. (Note that the TEST P program will be retained in the on-chip RAM at the $V_{LVI}$ voltage level.)

TYPICAL APPLICATION

A circuit that can be used to directly control a Darlington bipolar solenoid driver is shown in Figure 5. The MC680SP4L1 was chosen for the circuit because of its larger RAM area (112 bytes versus 64 bytes on other masked ROM devices). As shown in this figure, the port B (PBO) high current drive capability is used for this circuit. When the LVl capability is utilized, the solenoid immediately turns off with loss of power or brownout condition.
FIGURE 3 — MC6805P4 LVI Test Circuit Schematic Diagram

* If crystal option is used, delete 15 k resistor and connect crystal per data sheet.

FIGURE 4 — Test Pattern Routine (TEST P)
FIGURE 5 — MC6805P4 With Darlington Connected Solenoid Driver
Software to control the solenoid (via port B, PB0) is shown in Figure 6. This software routine may be entered into on-chip RAM via the RS-232-C terminal with the MC6805P4L1 in the monitor mode. The on-time of the solenoid (Allied Controls #306X-32) is set by the value entered (by the RS-232-C terminal) in RAM location $04F. The on/off times for this particular solenoid and software timing loops are shown below.

$04F = 04; 9.6 ms/9.6 ms
= 0F; 36 ms/36 ms
= 4F; 188 ms/188 ms
= FF; 620 ms/620 ms

The minimum on/off time of the solenoid shown in Figure 5 is 13 milliseconds. The 9.6 millisecond value (RAM $04F = 04 in Figure 6) only caused the solenoid to chatter; therefore, a higher hexadecimal value may be required in location $04F.

**MONITOR SOFTWARE**

A listing of the monitor (and self-check) program in the MC6805P4L1 (P1) is attached to this application note. This monitor program permits selection (via switch positions) of four baud rates which allows the user to enter and execute small software programs directly from on-chip RAM, as is done in this application note.

**PORT I/O CHARACTERISTICS WITH LVI OPTION**

The device operates successfully down to the $V_{LVI}$ threshold. During a lowering of $V_{CC}$ voltage, due to a brownout condition of electrical power or other reason, the LVI option provides the user with reduced drive capability but a stable port configuration. A discussion of these characteristics is provided below.

The $V_{LVI}$ threshold range may be between +2.7 V and +4.7 Vdc. The exact level could be determined as described above for the LVI Testing. Between the range of +4.7 Vdc and the actual $V_{LVI}$ voltage, each I/O port configuration remains as programmed. However, the drive capabilities (as a percentage of the specified port dc electrical characteristics) between the 4.7 Vdc and the actual $V_{LVI}$ voltage are as follows:

1. $I_{LOAD}$ Sink — 100% of specified current sinking capability for ports A and C, and 50% of specified sinking capability for port B.

2. $I_{LOAD}$ Source — 0% of specified drive current. That is, do not depend upon the M6805 HMOS Family MCU to source current below the lower $V_{CC}$ limit of +4.75 Vdc.

---

![FIGURE 6 - Software Listing for Controlling Solenoid via PB0](image-url)
MC6805P4 ROM PATTERN

This ROM contains a customer program, the standby monitor and a new version of the selfcheck. The customer ROM occupies $80-$FF and $3CO-$5Da. This leaves $5D9-$783 for the monitor. Most of the monitor commands have been deleted to make it fit within this area. The selfcheck is different from the P2 pattern since there is more RAM on this part. The old P2 selfcheck was very tightly coded and depended on there being exactly 64 bytes of RAM. The P4 has 112 bytes of RAM and requires a new test. This messes things up somewhat since the same good test cannot be applied to the P4.

The monitor is selected when bit 7 of port B is high during reset. Otherwise, the customer program will be selected. The monitor mode is an enhanced version of the original P2 version.

Serial I/O is handled on port C. Bit 3 is the serial output line and bit 2 is the serial input line. The lower two bits of port C select the baud rate for all serial I/O.

<table>
<thead>
<tr>
<th>C1</th>
<th>C0</th>
<th>BIT RATE/SEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1200</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4800</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>9600</td>
</tr>
</tbody>
</table>

ED Rupp May 11, 1981

I/O REGISTER ADDRESSES

<table>
<thead>
<tr>
<th>ADDR</th>
<th>NAME</th>
<th>EQU</th>
</tr>
</thead>
<tbody>
<tr>
<td>$000</td>
<td>PORTA</td>
<td>EQU 000</td>
</tr>
<tr>
<td>$001</td>
<td>PORTB</td>
<td>EQU 001</td>
</tr>
<tr>
<td>$002</td>
<td>PORTC</td>
<td>EQU 002</td>
</tr>
<tr>
<td>$003</td>
<td>DDR</td>
<td>EQU 4</td>
</tr>
<tr>
<td>$008</td>
<td>TIMER</td>
<td>EQU 008</td>
</tr>
<tr>
<td>$009</td>
<td>TCR</td>
<td>EQU 009</td>
</tr>
<tr>
<td>$010</td>
<td>RAM</td>
<td>EQU 010</td>
</tr>
<tr>
<td>$080</td>
<td>ZROM</td>
<td>EQU 080</td>
</tr>
<tr>
<td>$300</td>
<td>ROM</td>
<td>EQU 300</td>
</tr>
<tr>
<td>$800</td>
<td>MEMSIZ</td>
<td>EQU 800</td>
</tr>
<tr>
<td>$5D9</td>
<td>MONST</td>
<td>EQU 5D9</td>
</tr>
<tr>
<td>$5B3</td>
<td>CENTRY</td>
<td>EQU 5B3</td>
</tr>
</tbody>
</table>

CHARACTER CONSTANTS

<table>
<thead>
<tr>
<th>CODE</th>
<th>NAME</th>
<th>EQU</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>CR</td>
<td>EQU $0D</td>
</tr>
<tr>
<td>0A</td>
<td>LF</td>
<td>EQU $0A</td>
</tr>
<tr>
<td>20</td>
<td>BL</td>
<td>EQU $20</td>
</tr>
<tr>
<td>07</td>
<td>BEEP</td>
<td>EQU $07</td>
</tr>
<tr>
<td>00</td>
<td>EOS</td>
<td>EQU $00</td>
</tr>
</tbody>
</table>
ROM MONITOR FOR THE 6805P4

THE MONITOR HAS THE FOLLOWING COMMANDS:

M -- MEMORY EXAMINE/CHANGE.
   TYPE M AAA TO BEGIN,
   THEN TYPE: . -- TO RE-EXAMINE CURRENT
   E -- TO EXAMINE PREVIOUS
   CR -- TO EXAMINE NEXT
   DD -- NEW DATA
   ANYTHING ELSE EXITS MEMORY COMMAND.

E -- EXECUTE FROM ADDRESS. FORMAT IS
   E AAA. AAA IS ANY VALID MEMORY ADDRESS.

SPECIAL EQUATES

0000 00 2E  PROMPT EQU '.'  PROMPT CHARACTER
0000 00 0D  FWD  EQU CR    GO TO NEXT BYTE
0000 00 5E  BACK EQU '@'   GO TO PREVIOUS BYTE
0000 00 2E  SAME EQU '.'    RE-EXAMINE SAME BYTE

0000 00 7F  INITSP  EQU $7F  INITIAL STACK POINTER VALUE
0000 00 7A  STACK  EQU INITSP-5  TOP OF STACK

RAM VARIABLES

0000

0010  GET  RMB  4  NO-MANS LAND, SEE PICK AND DROP SUBROUTINES
0014  ATEMP RMB  1  ACCA TEMP FOR GETC,PUTC
0015  XTEMP RMB  1  IX TEMP FOR GETC,PUTC
0016  CHAR RMB  1  CURRENT INPUT/OUTPUT CHARACTER
0017  COUNT RMB  1  NUMBER OF BITS LEFT TO GET/SEND

0018

ORG MONST

MAIN --- PRINT PROMPT AND DECODE COMMANDS

0509 CD 06 B1  MAIN JSR CRLF  GO TO NEXT LINE
050C A6 2E     LDA  #PROMPT
050E CD 07 28   JSR  PUTC  PRINT THE PROMPT
0511 CD 06 F6   JSR  GETC  GET THE COMMAND CHARACTER
0514 A4 7F      AND  #1111111  MASK PARITY
0516 CD 06 BE   JSR  PUTS  PRINT SPACE (WON'T DESTROY A)
0519 A1 45      CMP  '#E' EXECUTE
051B 27 0B      BEQ  EXEC
051D A1 4D      CMP  '#M' MEMORY
051F 27 17      BEQ  MEMORY
0523 A6 3F      LDA  '#?' NONE OF THE ABOVE
0525 CD 07 28   JSR  PUTC
0528 20 E1      BRA  MAIN LOOP AROUND

272
* EXEC --- EXECUTE FROM GIVEN ADDRESS

05FB CD 06 D8 EXEC JSR GETNYB GET HIGH NYBBLE
05FB 25 DC  BCS MAIN BAD DIGIT
05FD 97  TAX  SAVE FOR A SECOND
05F2 CD 06 C7  JSR GETBYT NOW THE LOW BYTE
0601 25 D6  BCS MAIN BAD ADDRESS
0603 B7 7F  STA STACK+5 PROGRAM COUNTER LOW
0605 BF 7E  STX STACK+4 PROGRAM COUNTER HIGH
0607 80  RTI

* MEMORY --- MEMORY EXAMINE/CHANGE

0608 CD 06 D8  MEMORY JSR GETNYB BUILD ADDRESS
060B 25 CC  BCS MAIN BAD HEX CHARACTER
060D B7 11  STA GET+1
060F CD 06 C7  JSR GETNYT
0612 25 C5  BCS MAIN BAD HEX CHARACTER
0614 B7 12  STA GET+2 ADDRESS IS NOW IN GET+1&2
0616 CD 06 B1  MEM2 JSR CRLF BEGIN NEW LINE
0619 B6 11  LDA GET+1 PRINT CURRENT LOCATION
061B CD 06 A0  JSR PUTNYB
061E B6 12  LDA GET+2
0620 CD 06 93  JSR PUTBYT
0623 CD 06 BE  JSR PUTS A BLANK, THEN
0626 AD 33  BSR PICK GET THAT BYTE
0628 CD 06 93  JSR PUTBYT AND PRINT IT
062B CD 06 BE  JSR PUTS ANOTHER BLANK,
062E CD 06 C7  JSR GETBYT TRY TO GET A BYTE
0631 25 06  BCS MEM3 MIGHT BE A SPECIAL CHARACTER
0633 AD 2C  BSR DROP OTHERWISE, PUT IT AND CONTINUE
0635 AD 3A  MEM4 BSR BUMP GO TO NEXT ADDRESS
0637 20 DD  BRA MEM2 AND REPEAT
0639 A1 2E  MEM3 CMP #SAME RE-EXAMINE SAME?
063B 27 D9  BEQ MEM2 YES, RETURN WITHOUT BUMPING
063D A1 0D  CMP #FWD GO TO NEXT?
063F 27 F4  BEQ MEM4 YES, BUMP THEN LOOP
0641 A1 5E  CMP #BACK GO BACK ONE BYTE?
0643 26 94  BNE MAIN NO, EXIT MEMORY COMMAND
0645 3A 12  DEC GET+2 DECREMENT LOW BYTE
0647 B6 12  LDA GET+2 CHECK FOR UNDERFLOW
0649 A1 FF  CMP #$FF
064B 26 C9  BNE MEM2 NO UNDERFLOW
064D 3A 11  DEC GET+1
064F B6 11  LDA GET+1 SAME FOR HIGH NYBBLE
0651 A1 FF  CMP #$FF
0653 26 C1  BNE MEM2 TO WRAP AROUND
0655 A6 07  LDA #$7 HIGHEST ADDRESS IS $7FF
0657 B7 11  STA GET+1
0659 20 BB  BRA MEM2

* UTILITIES

* PICK --- GET BYTE FROM ANYWHERE IN MEMORY
* THIS IS A HORRIBLE ROUTINE (NOT MERELY
* SELF-MODIFYING, BUT SELF-CREATING)

273
* GET+1&2 POINT TO ADDRESS TO READ,
  * BYTE IS RETURNED IN A
  * X IS UNCHANGED AT EXIT

065B BF 15  PICK  STX  XTEMP  SAVE X
065D AE D6  LDX  #$D6  D6=LDA 2-BYTE INDEXED
065F 20 04  BRA  COMMON

* DROP --- PUT BYTE TO ANY MEMORY LOCATION.
  * HAS THE SAME UNDESIRABLE PROPERTIES
  * AS PICK
  * A HAS BYTE TO STORE, AND GET+1&2 POINTS
  * TO LOCATION TO STORE
  * A AND X UNCHANGED AT EXIT

0661 BF 15  DROP  STX  XTEMP  SAVE X
0663 AE D7  LDX  #$D7  D7=STA 2-BYTE INDEXED

0665 BF 10  COMMON  STX  GET  PUT OPCODE IN PLACE
0667 AE 81  LDX  #$81  81=RTS
0669 BF 13  STX  GET+3  NOW THE RETURN
066B 5F  CLRX  WE WANT ZERO OFFSET
066C BD 10  JSR  GET  EXECUTE THIS MESS
066E BE 15  LDX  XTEMP  RESTORE X
0670 81  RTS  AND EXIT

* BUMP --- ADD ONE TO CURRENT MEMORY POINTER
  * A AND X UNCHANGED

0671 3C 12  BUMP  INC  GET+2  INCREMENT LOW BYTE
0673 26 02  BNE  BUMP2  NON-ZERO MEANS NO CARRY
0675 3C 11  INC  GET+1  INCREMENT HIGH NYBBLE
0677 81  BUMP2  RTS

* OUT3HS --- PRINT WORD POINTED TO AS AN ADDRESS, BUMP POINTER
  * X IS UNCHANGED AT EXIT

0678 AD E1  OUT3HS  BSR  PICK  GET HIGH NYBBLE
067A A4 07  AND  #$111  MASK UNUSED BITS
067C AD 22  BSR  PUTNYB  AND PRINT IT
067E AD F1  BSR  BUMP  GO TO NEXT ADDRESS

* OUT2HS --- PRINT BYTE POINTED TO, THEN A SPACE. BUMP POINTER
  * X IS UNCHANGED AT EXIT

0680 AD D9  OUT2HS  BSR  PICK  GET THE BYTE
0682 B7 10  STA  GET  SAVE A
0684 44  LSRA
0685 44  LSRA
0686 44  LSRA
0687 44  LSRA
0688 AD 16  BSR  PUTNYB
068A B6 10  LDA  GET
068C AD 12  BSR  PUTNYB
06BE AD E1  BSR  BUMP  GO TO NEXT
0690 AD 2C  BSR  PUTS  FINISH UP WITH A BLANK
0692 81  RTS

*  PUTBYT  ---  PRINT A IN HEX  
*  A AND X UNCHANGED  

0693 B7 10  PUTBYT  STA  GET  SAVE A
0695 44  LSRA
0696 44  LSRA
0697 44  LSRA
0698 44  LSRA  SHIFT HIGH NYBBLE DOWN
0699 AD 05  BSR  PUTNYB  PRINT IT
069B B6 10  LDA  GET
069D AD 01  BSR  PUTNYB  PRINT LOW NYBBLE
069F 81  RTS

*  PUTNYB  ---  PRINT LOWER NYBBLE OF A IN HEX  
*  A AND X UNCHANGED, HIGH NYBBLE  
*  OF A IS IGNORED.  

06A0 B7 13  PUTNYB  STA  GET+3  SAVE A IN YET ANOTHER TEMP  
06A2 A4 0F  AND  #$F  MASK OFF HIGH NYBBLE  
06A4 AB 30  ADD  #'0  ADD ASCII ZERO  
06A6 A1 39  CMP  #$9  CHECK FOR A-F  
06A8 23 02  BLS  PUTNY2  
06AA AB 07  ADD  #'A-'9-1  ADJUSTMENT FOR HEX A-F  
06AC AD 7A  PUTNY2  BSR  PUTC  
06AE B6 13  LDA  GET+3  RESTORE A
0680 81  RTS

*  CRLF  ---  PRINT CARRIAGE RETURN, LINE FEED  
*  A AND X UNCHANGED  

0681 B7 10  CRLF  STA  GET  SAVE
0683 A6 0D  LDA  #$CR  
0685 AD 71  BSR  PUTC  
0687 A6 0A  LDA  #$LF  
0689 AD 6D  BSR  PUTC  
068B B6 10  LDA  GET  RESTORE
068D 81  RTS

*  PUTS  ---  PRINT A BLANK (SPACE)  
*  A AND X UNCHANGED  

06BE B7 10  PUTS  STA  GET  SAVE
06C0 A6 20  LDA  #$BL  
06C2 AD 64  BSR  PUTC  
06C4 B6 10  LDA  GET  RESTORE
06C6 81  RTS

*  GETBYT  ---  GET A HEX BYTE FROM TERMINAL  
*  A GETS THE BYTE TYPED IF IT WAS A VALID HEX NUMBER,  
*  OTHERWISE A GETS THE LAST CHARACTER TYPED. THE C-BIT IS  
*  SET ON NON-HEX CHARACTERS; CLEARED OTHERWISE. X  
*  UNCHANGED IN ANY CASE.
PAGE 006 MONIT .P4:1

06C7 AD 0F GETBYT BSR GETNYB BUILD BYTE FROM 2 NYBBLES
06C9 25 OC BCS NOBYT BAD CHARACTER IN INPUT
06CC 48 ASLA
06CD 48 ASLA
06CE 48 ASLA
06CF B7 10 ASLA
06D1 AD 05 STA GET SAVE IT
06D3 25 02 BCS NOBYT BAD CHARACTER NOW
06D5 BB 10 ADD GET C-BIT CLEARED
06D7 81 NOBYT RTS

* * GETNYB --- GET HEX NYBBLE FROM TERMINAL
* * A GETS THE NYBBLE TYPED IF IT WAS IN THE RANGE 0-F,
* * OTHERWISE A GETS THE CHARACTER TYPED. THE C-BIT IS SET
* * ON NON-HEX CHARACTERS; CLEARED OTHERWISE. X IS
* * UNCHANGED.

06D8 AD 1C GETNYB BSR GETC GET THE CHARACTER
06DA A4 7F AND #$11111111 MASK PARITY
06DC B7 13 STA GET+3 SAVE IT JUST IN CASE
06DE A0 30 SUB #'0' SUBTRACT ASCII ZERO
06ED 2B 10 BMI NOTHEX WAS LESS THAN '0'
06E2 A1 09 CMP #$9
06E4 23 0A BLS GOTIT
06E6 A0 07 SUB '#A-'9-1 FUNNY ADJUSTMENT
06EB A1 0F CMP #$F TOO BIG?
06EA 22 06 BHI NOTHEX WAS GREATER THAN 'F'
06EC A1 09 CMP #$9 CHECK BETWEEN 9 AND A
06EE 23 02 BLS NOTHEX
06F0 98 GOTIT CLC C=0 MEANS GOOD HEX CHAR
06F1 81 RTS
06F2 B6 13 NOTHEX LDA GET+3 GET SAVED CHARACTER
06F4 99 SEC RTS RETURN WITH ERROR
06F5 81

* * S E R I A L I / O R O U T I N E S
* * DEFINITION OF SERIAL I/O LINES
* *
06F6 00 02 PUT EQU PORTC SERIAL I/O PORT
06F6 00 02 IN EQU 2 SERIAL INPUT LINE#
06F6 00 03 OUT EQU 3 SERIAL OUTPUT LINE#

* * GETC --- GET A CHARACTER FROM THE TERMINAL
* * A GETS THE CHARACTER TYPED, X IS UNCHANGED.
* * INTERRUPTS ARE MASKED ON ENTRY
* * AND UNMASKED ON EXIT.
* *
06F6 BF 15 GETC STX XTEMP SAVE X
06FA B7 17 STA COUNT
06FC 04 02 FD GETC4 BRSET IN,PUT,GETC4 WAIT FOR HILO TRANSITION

* * DELAY 1/2 BIT TIME

276
PAGE 007 MONIT .P4:1

* LDA PUT
06FF B6 02 AND #11 GET CURRENT BAUD RATE
0701 A4 03 TAX
0703 97 DELAYS,X GET LOOP CONSTANT
0704 DE 07 6D GETC3 LDA #4
0707 A6 04 GETC2 DECA
0709 4A GETC2 BNE GETC2
070A 26 FD LOOP PAD
070C 4D DECX
070D 5A BNE GETC3 MAJOR LOOP TEST
070E 26 F7 *
* NOW WE SHOULD BE IN THE MIDDLE OF THE START BIT *
* *
0710 04 02 E9 BRESET IN,PUT,GETC4 FALSE START BIT TEST
0713 7D TST ,X MORE TIMING DELAYS
0714 7D TST ,X *
* MAIN LOOP FOR GETC *
* *
0715 AD 3E GETC7 BSR DELAY COMMON DELAY ROUTINE
0717 05 02 00 BRCLR IN,PUT,GETC6 TEST INPUT AND SET C-BIT
071A 7D GETC6 TST ,X TIMING EQUALIZER
071B 36 16 ROR CHAR ADD THIS BIT TO THE BYTE
071D 3A 17 DEC COUNT
071F 26 F4 BNE GETC3 STILL MORE BITS TO GET(SEE?)
0721 AD 32 BSR DELAY WAIT OUT THE 9TH BIT
0723 B6 16 LOA CHAR GET ASSEMBLED BYTE
0725 BE 15 LDX XTEMP RESTORE X
0727 81 RTS AND RETURN
* * PUTC --- PRINT A ON THE TERMINAL *
* X AND A UNCHANGED *
*SAME GAMES ARE PLAYED WITH THE I-BIT AS IN GETC *
* *
0728 B7 16 PUTC STA CHAR
072A B7 14 STA ATEMP SAVE IT IN BOTH PLACES
072C BF 15 STX XTEMP DON'T FORGET ABOUT X
072E A6 09 LOA #9 GOING TO PUT OUT
0730 B7 17 STA COUNT 9 BITS THIS TIME
0732 5F CLRX FOR VERY OBSCURE REASONS
0733 98 CLC THIS IS THE START BIT
0734 9B, SEI MASK INTERRUPTS WHILE SENDING
0735 20 02 BRA PUTC2 JUMP IN THE MIDDLE OF THINGS *
* *
0737 36 16 PUTC5 ROR CHAR GET NEXT BIT FROM MEMORY
0739 24 04 PUTC2 BCC PUTC3 NOW SET OR CLEAR PORT BIT
073B 16 02 BSET OUT,PUT
073D 20 04 BRA PUTC4
073F 17 02 PUTC3 BCLR OUT,PUT
0741 20 00 BRA PUTC4 EQUALIZE TIMING AGAIN
0743 DD 07 55 PUTC4 JSR DELAY,X MUST BE 2-BYTE INDEXED JSR
THIS IS WHY X MUST BE ZERO

0746 3A 17  DEC  COUNT
0748 26 ED  BNE  PUTC5  STILL MORE BITS
074A 14 02  BSET  IN,PUT  7 CYCLE DELAY
074C 16 02  BSET  OUT,PUT  SEND STOP BIT

074E AD 05  BSR  DELAY  DELAY FOR THE STOP BIT
0750 BE 15  LDX  XTEMP  RESTORE X AND
0752 B6 14  LDA  ATEMP  OF COURSE A
0754 81  RTS

* DELAY --- PRECISE DELAY FOR GETC/PUTC

0755 B6 02  DELAY  LDA  PUT  FIRST, FIND OUT
0757 A4 03  AND  #$11  WHAT THE BAUD RATE IS
0759 97  TAX
075A DE 07 6D  LDX  DELAYS,X  LOOP CONSTANT FROM TABLE
075C A6 F8  LDA  #$F8  FUNNY ADJUSTMENT FOR SUBROUTINE OVERHEAD
075E AB 09  DEL3  ADD  #$09
0761 4A  DEL2  DECA
0762 26 FD  BNE  DEL2
0764 5D  TSTX  LOOP PADDING
0765 14 02  BSET  IN,PUT  DITTO
0767 5A  DECX
0768 26 F5  BNE  DEL3  MAIN LOOP
076A A6 00  LDA  #0  FINAL TINY DELAY
076C 81  RTS  WITH X STILL EQUAL TO ZERO

* DELAYS FOR BAUD RATE CALCULATION

* THIS TABLE MUST NOT BE PUT ON PAGE ZERO SINCE
* THE ACCESSING MUST TAKE 6 CYCLES.

076D 20  DELAYS  FCB  32  300 BAUD
076E 08  FCB  8  1200 BAUD
076F 02  FCB  2  4800 BAUD
0770 01  FCB  1  9600 BAUD

* RESET --- POWER ON RESET ROUTINE

* CHECK FOR CUSTOMER OR MONITOR MODE AND BRANCH
* ACCORDINGLY.

0771 4E 01 03  BRSET  7,PORTB,MONIT
0774 CC 05  B3  JMP  CENTRY CUSTOMER PROGRAM ENTRY POINT
0777  A6 08  MONIT
0777 A6 08  LDA  #$1000  SETUP PORT FOR SERIAL IO
0779 B7 02  STA  PUT  SET OUTPUT TO MARK LEVEL
077B B7 06  STA  PUT+DDR SET DDR TO HAVE ONE OUTPUT
077D B6 02  LDA  PUT
077F 83  SWI  GO TO MONITOR ROUTINE
0780 20 EF  BRA  RESET LOOP AROUND

*******************************************************************************

* 6805 SELFTEST
GENERAL:

SELFTEST PERFORMS THE FOLLOWING TESTS:

- PORTC -- TEST PORTC FOR GOOD INPUT
- I/O -- PORT A AND B FOR INPUT AND OUTPUT
- RAM -- WALKING BIT MEMORY TEST
- ROM -- EXCLUSIVE OR WITH ODD 1'S PARITY RESULT
- INTERRUPT -- INT AND TIMER INTERRUPTS

PORTC TEST IS DONE ONLY ONCE JUST AFTER RESET. THE OTHERS
ARE REPEATED AS LONG AS NO ERRORS ARE FOUND.

THE TEST STATUS IS AVAILABLE ON PORTC. AFTER THE PORTC
TEST, PORTC IS USED AS AN OUTPUT TO INDICATE WHICH TEST IS
RUNNING. IF ANY TEST FAILS, THE OUTPUT LINES REMAIN
STABLE AND THE LOWER 2 BITS OF PORTC INDICATE THE TEST
THAT FAILED (ASSUMING PORTC WORKS!). SINCE THERE ARE ONLY
4 'CHECKPOINTS' IN THE MAIN LOOP, IT IS NOT POSSIBLE TO
ALWAYS DETERMINE EXACTLY WHAT IS WRONG WITH THE PART.
HERE ARE THE PROBABLE ERRORS IF THE TEST HAS STOPPED:

- B1 B0 PROBLEM
- 0 0 INTERRUPT FAILURE
- 0 1 BAD PORTA OR PORTB
- 1 0 BAD RAM
- 1 1 BAD ROM

THE TIME REQUIRED FOR ONE CYCLE OF THE PROGRAM IS
APPROXIMATELY 800 MILISECONDS. AFTER EACH TEST, PORTC IS
INCREMENTED. THEREFORE, BIT 1 SHOULD APPEAR TO OSCILLATE
WITH A PERIOD OF ABOUT 800MS. IT MAY BE DIFFICULT TO
DISCERN THIS SINCE THE DUTY CYCLE ON BIT 1 IS NOT ANYWHERE
NEAR A SQUARE WAVE. BIT 2 AND BIT 3 WILL HOWEVER HAVE
SQUARE WAVE OUTPUTS. BIT 2'S PERIOD WILL BE TWICE AS LONG
AS BIT 1, AND BIT 3 SHOULD BLINK AT ABOUT 0.3HZ.

THROUGHOUT THIS PROGRAM, IT MAY SEEM THAT THE PROGRAMMERS
HAVE GONE OUT OF THEIR WAY TO USE STRANGE INSTRUCTIONS AND
UNUSUAL TECHNIQUES. THIS IS TRUE. THE PURPOSE OF ALL
THIS IS TO 1) REDUCE THE PROGRAM SIZE AS MUCH AS POSSIBLE
AND 2) TO EXECUTE A BROAD RANGE OF INSTRUCTION TYPES AND
ADDRESSING MODES.

0782

PAG

EQUATES

0782 00 00

ANY EQU 0 RANDOM BIT IN A BYTE

0782 00 3F RAMSUB EQU $3F START OF BUILT SUBROUTINE
0782 00 20 LOC EQU $20 USED IN INTERRUPT TEST
0782 00 7C STACKA EQU $7C A WHEN STACKED BY INTERRUPT
0782 00 7D STACKX EQU $7D X THE SAME
ORG MEMSIZ-116-B

BEGIN OF SELF TEST

START RSP RESET JUST IN CASE (AVOID STACK INIT PROBLEM)
COM PORTC SHOULD HAVE READ $FF
BNE * PORTC BAD ON INPUT

COM SET PORTC TO ZERO

BSET ANY, PORTC+DDR AND PROGRAM DATA DIRECTION

MAIN LOOP (REPEATED)

CLR TIMER+1 RESET TIMER INTERRUPTS

SWI PASSED FIRST TEST

INPUT/OUTPUT PORT TESTS

IOTST LDX #PORTB POINT TO PORTB FIRST
IOTST2 LDA #$6F
BSR IOSUB RETURNS WITH A=$54
ADC #$0F-$54-1 A IS SET TO $0F
BSR IOSUB AGAIN WITH NYBBLES REVERSED
DECL
BEQ IOTST2 AGAIN FOR PORTA
CLRA CLEAR A FOR NEXT TEST

SWI PASSED I/O TEST

RAM TEST

ENTER WITH C=1, A=0.

THIS TEST IS MODIFIED FROM THE ORIGINAL P2 RAM TEST. A NEW TEST IS NECESSARY BECAUSE THERE ARE 112 BYTES OF RAM ON THE P4. THE OLD RAM TEST WORKED BECAUSE THE TEST SEQUENCE WAS 9 BYTES LONG, AND THERE WERE 64 BYTES OF RAM. 64 MOD 9 IS 1, SO A SINGLE ROTATE WAS SUFFICIENT TO RESTORE THE SEED. 112 MOD 9 IS 4 WHICH REQUIRE 4 ROTATES TO GET BACK TO THE INITIAL PATTERN. THIS EXCEEDS THE AVAILABLE ROM.

THE NEW RAM TEST IS MUCH SIMPLER, BUT WILL TEST ALL THE BYTES. THIS CHANGE AFFECTS THE SUBSEQUENT TESTS SINCE THEY MUST NOW INITIALIZE THEIR OWN VARIABLES INSTEAD OF RELYING ON THE RAM TEST TO DO IT FOR THEM.

RAM WILL BE ALL ZEROES AFTER THIS TEST.

LDX #RAM CLEAR OUT A BYTE
STA ,X
CMP ,X COMPARE WITH ACCUMULATOR VALUE
BNE * MEMORY MISMATCH
INC ,X BUMP BOTH VALUES
07A5 26 F9           BNE      RAM3
07A7 5C              INCX     ADVANCE TO NEXT RAM BYTE
07A8 2A F5           BPL      RAM2  CONTINUE TILL ALL TESTED

EXIT WITH A=0, X=$80

07AA 83              SWI      PASSED RAM TEST

ROM TEST

FORCE A RESULT OF $FF (1'S PARITY). THE CHECKSUM HAS BEEN
CHOSEN TO FORCE THIS RESULT IF THE ROM ITSELF IS GOOD.
ADDRESS AND DATA LINES STUCK HIGH, LOW OR TO EACH OTHER
ARE DETECTED WITH HIGH PROBABILITY.

FOR A DISCUSSION OF THE RELIABILITY OF THIS METHOD OF
TESTING THE ROM, SEE "MICROPROCESSOR BASED DESIGN" BY DR.
J. B. PEATMAN, PAGES 315-316.

ENTER WITH A=$00,X=$80, RAM=00

07AB

07AB BF 41           STX      RAMSUB+2
07AD AE C8           LDX      #$C8  =OR EXTENDED
07AF BF 3F           STX      RAMSUB
07B1 AE 81           LDX      #$81  =RTS
07B3 BF 42           STX      RAMSUB+3
07B5 BD 3F           JSR      RAMSUB
07B7 3C 41           INC      RAMSUB+2 ADVANCE TO NEXT ADDRESS
07B9 22 FA           BHI      SUM
07BB 3C 40           INC      RAMSUB+1
07BD 07 40 F5        BRCLR     3,RAMSUB+1, SUM LOOK FOR $8 IN RAMSUB+1
07C0 43              COMA     A SHOULD HAVE BEEN $FF,
07C1 26 FE           BNE      * HANG HERE FOR BAD ROM

07C3 83              SWI      PASSED ROM TEST

INTERRUPTS TEST

ENTER WITH: X=$81
A=$00
LOC=$80

INTERRUPT TEST ALLOWS INTERRUPTS LONG ENOUGH TO GET ONE
TIMER INTERRUPT AND ONE INT INTERRUPT. THE INTERRUPT
SERVICE ROUTINES SHIFT A BYTE IN MEMORY TO A KNOWN PATTERN
WHICH IS CHECKED AFTER THE INTERRUPTS SHOULD HAVE OCCURED.
FURTHER, THE A AND X REGISTER ARE COMPARED WITH WHAT WAS
STACKED DURING THE INTERRUPTS.

A TIMER INTERRUPT IS GUARANTEED PENDING BY ALLOWING ENOUGH
TIME TO ELAPSE DURING THE OTHER TESTS TO UNDERFLOW THE
COUNTER (EVEN WITH MAXIMUM PRE-SCALE). THE INT INTERRUPT
IS ALSO GUARANTEED SINCE THE INT LINE IS TIED TO THE PORTA
LINE WHICH WIGGLES UP AND DOWN DURING THE I/O TEST.
MONIT

PAGE 012 MONIT .P4:1

07C4 EF 20 BSET 7, LOC SET FLAG BIT IN LOC
07C6 9A CLI ALLOW INT AND TIMER INTERRUPT
07C7 9B SEI INTERRUPTS SHOULD BE DONE NOW
07C8 2C FE BMC * I-BIT NOT WORKING
07CA B0 7C SUB STACKA COMPARE A WITH STACKED VALUE
07CC 26 FE BNE * INTERRUPT FAILED
07CE B3 7D CFX STACKX ALSO CHECK X
07DF 26 FE BNE *

07D2 0B 20 FD BRCLR 5, LOC, * BIT SHOULD SHIFT HERE

END OF TESTS, DO IT AGAIN

* NOTE THE USE OF A 2-BYTE INDEXED JUMP WHICH FORCES A CARRY FROM THE LOWER BYTE IN COMPUTING THE JUMP ADDRESS.

07D5 DC 07 0A JMP LOOP-$81,X DO IT AGAIN

* I/O TEST SUBROUTINE

* ENTERED WITH X POINTING TO THE PORT TO TEST. A HAS THE DDR PATTERN TO USE.

* RETURNS WITH A=$54 AND C=1.

07D8 E7 04 IOSUB STA DDR,X SETUP DATA DIRECTION
07DA A6 55 LDA #$55 ALTERNATE 1'S AND 0'S
07DC F7 AGAIN STA ,X SAVE IT AND SEE
07DD F1 CMP ,X IF IT STAYS THERE
07DE 26 FE BNE * BAD I/O PORT(S)
07E0 4B LSLA SHIFT PATTERN TO $AA
07E1 2B P9 BMI AGAIN AND REPEAT TEST
07E3 A1 RTS

* INTERRUPT ROUTINES

* SOFTWARE INTERRUPT IS RESPONSIBLE FOR ADVANCING THE VALUE STORED IN PORTC.

07E4 3C 02 SOFT INC PORTC ADVANCE TO NEXT TEST
07E6 80 RTI AND EXIT

* INT AND TIMER INTERRUPTS

* SHIFT LOC ONE PLACE TO THE RIGHT.

07E7 1F 09 TIMEUP BCLR 7, TIMER+1 HANDLE INTERRUPT
07E9 37 20 INTR ASR LOC SHIFT FLAG AROUND
07EB 80 RTI

07EC 00 00 00 FCB 0,0,0 SPARE BYTES
07EF 9B CHKSUM FCB $9B ROM CHECKSUM (ODD 1'S PARITY)

* SELFTEST ALTERNATE INTERRUPT VECTORS

07F0 07 E7 FDB TIMEUP
07F2 07 E9 FDB INTR
07F4 07 E4 FDB SOFT
PAGE 013 MONIT .P4:1

07F6 07 84         FDB         START
07F8
*                  END RESET WAZU
*                  INTERRUPT VECTORS
07F8
*                  ORG         MEMSIZ-8 START OF VECTORS
07F8 05 6C         FDB         $56C CUSTOMER TIMER INTERRUPT
07FA 05 B3         FDB         $5B3 CUSTOMER INT VECTOR
07FC 05 D9         FDB         MAIN SWI TO MAIN ENTRY POINT OF MONITOR
07FE 07 71         FDB         RESET POWER ON VECTOR
INTRODUCTION

This application note covers some factors which should be considered when using on-chip analog-to-digital (A/D) converters. It is intended for the digital designer with little or no programming experience.

The task of converting analog signals into digital information is becoming easier using today’s technology. The MC6805R3 and MC6805S2 microcomputer (MCUs) have an on-chip 8-bit A/D converter with four user channels to help accomplish this task. These versatile MCU devices are easy to use and serve well in automotive, industrial, medical or environmental systems where analog information is monitored, controlled, or stored.

The merging of analog and digital circuits does present a problem to the designer who is most familiar and comfortable using only one of the two disciplines. The problem arises when the analog designer needs to write a program to process the analog information or the digital designer has to contend with the analog hardware characteristics with which he is not totally familiar.

This application note defines the terminology used when discussing A/D converters, describes the circuit elements pertinent to the user, illustrates a self-test hardware/software technique, and gives an example on how to manipulate the converted analog data from a temperature sensor. Program listings are provided at the end of this application note.

DEFINITION OF TERMS

For a better understanding of the characteristics of the M6805 Family 8-bit A/D converter, the key terms used to describe the capabilities and limitations of an A/D converter, are presented first. Also included in this presentation is a brief interpretation of the specifications in the current MC6805R2 data sheet.

Conversion Range

The voltage reference high (VRH) and the voltage reference low (VRL) pins establish the voltage range that is recognized by the converter. The sum of VRH and VRL should never exceed the VCC of the MCU. If VRH - VRL is below 4,000 volts, the specification accuracy can not be maintained. Operating below the minimum VRH limit of 4,000 volts may cause the converter to miss steps or lose accuracy.

Conversion Time

The A/D converter in an M6805 Family MCU is a successive approximation type. A hardware binary-search method is used to determine the unknown input voltage. This method significantly reduces the time (total of 30 machine cycles) needed to analyze the input voltage and generate a hexadecimal value accurately. An M6805 Family MCU with a full-speed, 4-MHz crystal has a 1-microsecond machine cycle, and thus a 30 microsecond conversion time.

Monotonicity/Missing Codes

Monotonicity is a function of the integral non-linearity of the converter. An A/D converter is monotonic if the output digital value always increases or remains the same when the analog input voltage is increasing. A non-monotonic A/D converter is one that skips steps or decreases in output digital value when the input is increasing.

Non-Linearity

Studying the relationship of the analog input voltage with respect to the digital value that is derived after a conversion on an ideal A/D converter, yields a linear graph similar to the one shown in Figure 1. Integral non-linearity is defined as the deviation from an ideal straight-line, transfer function. Differential non-linearity is the amount that a particular step differs from one least significant bit (LSB). An ideal A/D converter has equal steps, and thus no differential non-linearity. Total non-linearity error is then the sum of the differential and integral errors.

Quantizing Error

The converter’s inability to recognize voltage changes of less than one quantum step is referred to as the quantizing error. Since one increment or quantum is the smallest voltage
value that the A/D converter recognizes, there is an inherent offset designed into the circuit. The M6805 Family A/D converter minimizes the impact of the offset error by centering it about the ideal conversion point.

Figure 1 shows the quantizing error to be ±0.5 LSB. This offset enables the converter to detect input voltage changes ±0.5 LSB from the center of a step. Thus, the first step occurs at 0.5 LSB and all subsequent steps occur at one LSB increments from this first 0.5 LSB step (that is, 1.5 LSB, 2.5 LSB, etc.). Also, the last step ($F E$ to $F F$) is 1.5 LSB long because of this shift in the transfer function.

**Ratiometric Reading/Gain Error**

The gain error of the A/D comparator is the deviation from the ideal full scale input voltage and the full scale digital output value. The specification limit can be adjusted by trimming the $V_{RH}$ reference voltage when the input voltage is at the transfer point between $F E$ and $F F$ (5.090 volts for a nominal $V_{RH} = 5.120$ volts system) and looking for an output value of $F F$.

**Reference Voltage ($V_{RH}$/$V_{RL}$)**

The voltage reference high and voltage reference low pins provide the voltage limits to the digital-to-analog (D/A) converter resistor/capacitor chain in the A/D circuit. The nominal resistance value between these two pins is approximately 6 kilohms. The reference voltage variation during a conversion should not exceed 0.25 LSB.

**Resolution**

When referring to an analog voltage range, there are an infinite number of steps that can be realized. Since it is unrealistic to try and cover all the possible steps, a method of approximating the unknown value using a digital circuit was developed. If a data converter has 8-bits, the 8 refers to the number of steps in powers of two. For example, an 8-bit A/D converter has 256 steps (2^8).

The number of steps is used to determine the A/D resolution. Resolution is defined as the full scale input voltage divided by the total number of steps. This can be represented by the equation:

Resolution = \( \frac{V_{RH} - V_{RL}}{\text{Number of Steps}} \)

The $V_{RH}$ (voltage reference high) is the highest voltage that can be converted, and $V_{RL}$ (voltage reference low) is the low end of the conversion range.

This means that the converter recognizes input voltage changes no smaller than one incremental value. One increment can be expressed as a fraction of the full scale voltage or one least significant bit (LSB) of the N-bit digital word.

The 8-bit A/D converter implemented in an M6805 Family MCU uses a unipolar binary code. A value of $00$ represents an analog input voltage of 0.000 to 0.020 volts and a value of...
SFF represents an analog input voltage of 5.100 to 5.120 volts, provided $V_{RH} = 5.120$ volts and $V_{RL} = 0.000$ volts. A dollar sign ($) in front of a symbol identifies it in this document as a hexadecimal number (base 16 number).

Sample Time/Sample and Hold Capacitance

The sample time of the M6805 Family A/D converter is the first five machine cycles. This is the actual aperture window during which the selected analog channel coupler connects the sample and hold capacitor to the input voltage. The internal sample and hold capacitor charges for five machine cycles, and then holds that charge on the comparator input during the remaining 25 cycles of the conversion period.

The analog input voltage ideally should stay constant during the sample period. Any variation contributes to the conversion error. It is desirable to hold the input voltage within 0.5 LSB during the sample window. Under extreme operating conditions, the first conversion obtained after selecting a new channel may be less accurate than later conversions of the channel. Input channel changes can affect the stored sample and charge.

The input voltage on the channel coupler should not exceed $V_{RH}$ nor be less than $V_{RL}$. A voltage greater than $V_{RH}$ converts to SFF, but no overflow indication is provided. Since the maximum rate of change of the input signal is dependent upon the converter sample time, the maximum frequency of a sine wave input is: $f = \frac{1}{TS \times pi \cdot 2(N + 1)}$ where TS is the sample time and N is the number of bits. With this equation, the slew rate of the signal does not exceed 0.5 LSB during a sample of the input voltage.

Zero Input Reading/Offset Error

The input offset voltage of the A/D comparator causes a shift in its transfer function. The quantizing error inherent in the design of the comparator causes the 0.5 LSB shift in the step function. Since a 0.5 LSB error is also allowed in linearity error, an input of 0.000 volts can convert to $000$ or $001$.

THE M6805 FAMILY A/D CONVERTER

Figure 2 is a block diagram of the successive approximation A/D converter implemented in an M6805 Family MCU.

There are two unique registers which are addressed under program control. They are used to select a channel, start a conversion, and read the 8-bit result after a conversion has been completed.

There are four separate external input channels which can be individually coupled to the comparator input by writing to the three lower order bits in the A/D control register (ACR).

There are also four internal input channels which are coupled to the $V_{RH}/V_{RL}$ resistor chain and can be used as calibration references. Also shown in Figure 2 are the address locations for the external and internal channels.

The converter operates continuously producing a new result every 30 machine cycles. Bit 7 of the ACR flags the user when a conversion has been completed and the digital value of the analog input can be read from the A/D result register (ARR).

When the aperture window closes, the successive approximation register (SAR) addresses the D/A converter (DAC) with a binary word value of $S0$. Thus, only the most significant bit (MSB) of the word is set. This value causes the DAC to generate an analog voltage equal to the midway value of the conversion range.

At the comparator, the DAC output is compared with the input voltage stored in the sample and hold capacitor. The comparator then signals the SAR whether the DAC value is greater than the sampled value. If the sampled voltage is greater than the DAC voltage, the SAR binary value is increased to $SC0$ by setting the bit that is adjacent to the already set MSB. In turn, this increased value increases the DAC output voltage. If the input voltage is less than the DAC voltage, the SAR MSB is cleared and the bit adjacent to the MSB is set, resulting in less DAC voltage to the comparator. This binary-search method continues until all eight SAR bits have been determined.

At the end of the conversion, the SAR value is transferred to the A/D result register (ARR). This binary value is the approximate digital representation of the analog input within one LSB of error. A conversion complete flag is set in the ACR indicating to the user program that a new result may be retrieved.

The equivalent analog input circuit for one channel entering the multiplexer is illustrated in Figure 3. The worst case instantaneous current draw is at the beginning of the sample period. Since the RC time constant of the sample and hold circuit is approximately 500 nanoseconds, the five machine-cycle sample time is more than sufficient to charge the sample and hold capacitor.

The average analog channel input current is less than one microampere. Thus, no additional buffering of external transducer signals is required. The analog channel input impedance should not be greater than 1K ohms to prevent capacitor leakage currents from producing unwanted voltage drops.

A/D CONVERTER SELF-TEST

Designing a test capability into a system simplifies maintenance, decreases the need for costly test equipment in the field, and provides an excellent way to learn the system characteristics. Using a minimum amount of hardware an M6805 Family MCU can test its own A/D converter in less than one second for 8-bit accuracy. The objective of this example is to self-adjust the A/D converter under test for full scale error within two LSB and then to test for a correct conversion at the center of each step. The MCU used in this example is an MC68705R3.

The hardware schematic for the A/D converter self-test shown in Figure 4 consists of a voltage stimulus, a programmable voltage reference, and a program storage unit. A 12-bit resolution monolithic current output DAC (U2) presents the voltages to the MCU A/D converter input. An operational amplifier converts the DAC current output to a voltage and is fed into the device under test (DUT).

In order to download the test program to the MCU RAM, a 12-bit binary counter (U7) is used to address an EPROM (U3). Another 12-bit counter (U1) addresses the DAC in order to ramp the test voltages to the DUT. The two 12-bit counters are used to minimize the MCU pins needed for calibration in user systems. A variable-precision voltage reference (U3) and a quad analog switch (U6) are used to derive the voltage reference to the DUT.

BOOTSTRAP PROGRAM (BOOT)

To minimize the amount of ROM occupied by the self-test program, a 28-byte bootstrap loader routine is suggested. The bootstrap loader is a short program which is executed
FIGURE 2 — A/D Converter Block Diagram and Input Select Code

A/D INPUT MUX SELECTION

<table>
<thead>
<tr>
<th>Control Status Register</th>
<th>Input Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACR2</td>
<td>ACR1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Internal Calibration Levels

FIGURE 3 — Analog Channel Input Circuit
**FIGURE 4** — On-Chip A/D Converter Self-Test and Self-Calibration-Schematic Diagram
when a particular condition is met after the MCU comes out of reset. The bootstrap program loads a short test program from an external memory device into on-chip RAM and executes the program.

The condition selected in order to execute the bootstrap loader program is a logic low level on the interrupt pin when coming out of reset. When the interrupt pin has a logic high level, the user main program is entered. The address of the label "load" in the bootstrap program is used in the self-test program to jump back to load new routines. The bootstrap loader approach is not limited to the A/D converter self-test program. It can be used to examine or test other system features.

**SELF-TEST SOFTWARE (A/DST)**

The self-test software consists of three sequential loads. This requires the bootstrap loader to jump to RAM after downloading, execute a program from RAM, then return to the bootstrap loader to download the next routine. When the MCU comes out of reset, the interrupt pin is held low in order to enter the bootstrap loader routine.

The bootstrap program uses port B bit 1 to clear the counter and port B bit 0 to increment the count on a negative transition. The program is fed into port A and stored to RAM.

The index register is used as the RAM pointer and is tested for a negative value after each RAM byte load. Since the negative status bit in the condition code register is set when the accumulator holds a value between $80 and $FF, the loading routine ends when the index register reaches a value of $80. The Figure 5 flow chart illustrates the interaction between the bootstrap loader routine (BOOT) and the A/D converter self-test routines (A/DST).

**12-Bit DAC Calibration**

The first routine is used to calibrate the 12-bit DAC (U2) and the precision voltage reference (U5). The ideal \( V_{RH} \) selected is 5.120 volts for 0.020-volt steps. Therefore, the first step is to preset U5 with R5 to 5.160 volts (2 LSB above the ideal \( V_{RH} \)) while the MCU is held in reset.

The next step is to calibrate the 12-bit DAC zero output for 0.000 volts and the full scale output for 5.100 volts. This routine is executed only if port C bits 4 and 5 are held low. This is done by closing S1 (zero calibrate select) and S2 (full scale calibrate select) before bringing the MCU under test out of reset.

The 12-bit DAC zero offset adjustment is set by varying R6 while monitoring the DAC output for a value of 0.000 volts. Opening the bit 4 switch causes the DAC to be incremented to $FF0 and held there for the full-scale error adjustment provided the bit 5 switch is closed.

The DAC should then be calibrated to 5.100 volts by varying R15. Opening the bit 5 switch causes the calibration routine to terminate, and the second routine is then downloaded. The DAC calibration routine is normally done once after initial power-up and is skipped if the port C switches are open.

**On-Line A/D Calibration**

The second routine in the self-test exercise is the A/D converter automatic full-scale error adjust. The DAC is incremented to $FE8 (5.090 V), which is the A/D converter switch point between $FE and $FF. All analog switches on U6 are off at this time so the voltage reference to the DUT is 5.160 volts (\( V_{RH} \) ideal + 2 LSB). The A/D converter should output a value of $FE since the analog input switch point from $FE to $FF would be 5.140 volts using this reference value. Table 1 represents the voltage inputs that cause a switch in the digital output of the A/D converter when using a \( V_{RH} + V_{RL} \) of 5.120 V.

The analog switch is controlled by the four lower order bits of port C. As port C is incremented, it causes a decrease of 10 millivolts on the output of the voltage reference, U5. An A/D conversion is done after each port C increment to check for a value of $FF.

When the A/D converter recognizes the input voltage as an $FF, port C is left constant at that value for the next test. This self-adjusting routine compensates for the variation in full-scale error from one device to another and eliminates having to manually adjust each unit before test.

**A/D Linearity Test**

The third and last routine is the actual A/D converter voltage ramp test. The 12-bit DAC is cleared then ramped in 20-millivolt increments to check the center of each A/D step for a correct conversion. At the end of the 256 voltage steps, the program outputs a logic low level on the port B bit 2 to turn on an LED, indicating a good device.

If at any point during the 256 voltage steps a conversion is incorrect, testing is stopped and the program outputs a logic low on port B bit 3 to signal that the A/D converter failed the test. The A/D converter linearity is thus confirmed to be within one LSB (20 millivolts) of the proper reading.

**TEMPERATURE SENSOR CONVERSION**

Monitoring temperature for display or control is a key factor for energy conservation or process control in closed loop systems. The objective in this portion of the application note is to demonstrate how to manipulate the converted analog signal from a temperature sensor for display.

By monitoring the base-emitter voltage variations on the Motorola MTS-102 silicon temperature sensor, the MCU converts that analog information into an equivalent digital value in degrees fahrenheit and displays it on three 7-segment displays.

Many tasks can be performed after the conversion is complete. One typical application for this circuit is a ceiling fan control to increase room air flow for a period of time before turning on the main central air conditioning unit. If the temperature does not decrease after a timeout period, then the central air is turned on for a period of time.

The Figure 6 schematic shows how the sensor voltage is amplified to give 20-millivolt steps per degree fahrenheit. A dual differential amplifier (U2) buffers the sensor voltage and then inverts and amplifies the signal before entering the A/D converter on the MCU. An amplifier gain of eight produces 10 millivolts per degree fahrenheit, and a gain of 4.444 produces 10 millivolts per degree centigrade.

In order for the A/D converter to recognize one degree steps, the gain must be doubled. The maximum output voltage on the differential amplifier is approximately 3.8 volts using a \( V_{CC} \) of 5 volts. Therefore, the temperature sensing range is from -40 degrees to +140 degrees fahrenheit.

The Monsanto LED displays have a common anode and are driven directly by the MCU port B. To update the displays, port C bits 0, 1 and 2 enable each display sequentially every 2048 machine cycles. The regulated voltage from U1 is tied to the MCU \( V_{CC} \) as well as to the displays and is used ratiometrically with respect to the \( V_{RH} \) pin and the
Note:
Segments 1, 2, and 3 of the A/D test reside in RAM starting at address $40 after being loaded by the Boot program.

FIGURE 5 — Boot and A/D TST Program Flow Chart
(Sheet 1 of 2)
FIGURE 5 — Boot and A/D TST Program Flow Chart
(Sheet 2 of 2)
FIGURE 6 — Temperature Sensing A/D
Demonstration-Schematic Diagram
Sensor supply. Thus, any variation on the amplifiers also occurs on the \( V_{RH} \) pin.

Since the MC68705R3 has an EPROM erasing window, it is necessary to cover it after programming so that the A/D converter will maintain its 8-bit accuracy. It is also recommended that the MTS102 sensor leads be moisture-sealed with epoxy. About 12 to 18 inches of twisted, stranded, 22-gauge wire was used to connect the sensor to the amplifier input.

The Figure 7 flow chart describes the sequence of events in the temperature sensor program (TMPNS). After port initialization, the timer is enabled to periodically update the 7-segment displays. An A/D conversion is performed and the result is converted to binary-coded-decimal (BCD) format for the display routine.

Zero degrees fahrenheit is equal to a conversion of $30$ so the A/D conversion result must be offset before converting to a BCD number. Since the displays are updated during a timer interrupt, the MCU goes into a wait loop before doing another A/D conversion. This is a good entry point for appending new tasks to the basic program such as time of day or controlling a heater or air conditioner.

Calibration of the temperature sensor is performed by varying the 50-kilohm potentiometer (R1) on the differential amplifier for a reading of 32 degrees after a piece of ice has been placed on the sensor for approximately one minute. Refer to the MTS-102 specification for further details.

**SUMMARY**

The terminology used to describe an A/D converter characteristics has been discussed and related to the electrical characteristics in the M6805 Family A/D converter specifications. A circuit and program which tests the on-chip A/D converter was also discussed to familiarize the reader with a method of calibrating the A/D converter. A typical temperature sensor application was covered with the intent of showing the sequence of events that take place when converting the A/D result to a displayable value.

**TABLE 1** — Voltage Versus Output for 8-Bit A/D Converter

<table>
<thead>
<tr>
<th>Hi/Lo</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0°</td>
<td>0.010</td>
<td>0.330</td>
<td>0.650</td>
<td>0.970</td>
<td>1.290</td>
<td>1.610</td>
<td>1.930</td>
<td>2.250</td>
<td>2.570</td>
<td>2.890</td>
<td>3.210</td>
<td>3.530</td>
<td>3.850</td>
<td>4.170</td>
<td>4.490</td>
</tr>
<tr>
<td>1</td>
<td>0.030</td>
<td>0.350</td>
<td>0.670</td>
<td>0.990</td>
<td>1.310</td>
<td>1.630</td>
<td>1.950</td>
<td>2.270</td>
<td>2.590</td>
<td>2.910</td>
<td>3.230</td>
<td>3.550</td>
<td>3.870</td>
<td>4.190</td>
<td>4.510</td>
</tr>
<tr>
<td>2</td>
<td>0.050</td>
<td>0.370</td>
<td>0.690</td>
<td>1.010</td>
<td>1.330</td>
<td>1.650</td>
<td>1.970</td>
<td>2.290</td>
<td>2.610</td>
<td>2.930</td>
<td>3.250</td>
<td>3.570</td>
<td>3.890</td>
<td>4.210</td>
<td>4.530</td>
</tr>
<tr>
<td>3</td>
<td>0.070</td>
<td>0.390</td>
<td>0.710</td>
<td>1.030</td>
<td>1.350</td>
<td>1.670</td>
<td>1.990</td>
<td>2.310</td>
<td>2.630</td>
<td>2.950</td>
<td>3.270</td>
<td>3.590</td>
<td>3.910</td>
<td>4.230</td>
<td>4.550</td>
</tr>
<tr>
<td>4</td>
<td>0.090</td>
<td>0.410</td>
<td>0.730</td>
<td>1.050</td>
<td>1.370</td>
<td>1.690</td>
<td>2.010</td>
<td>2.330</td>
<td>2.650</td>
<td>2.970</td>
<td>3.290</td>
<td>3.610</td>
<td>3.930</td>
<td>4.250</td>
<td>4.570</td>
</tr>
<tr>
<td>5</td>
<td>0.110</td>
<td>0.430</td>
<td>0.750</td>
<td>1.070</td>
<td>1.390</td>
<td>1.710</td>
<td>2.030</td>
<td>2.350</td>
<td>2.670</td>
<td>2.990</td>
<td>3.310</td>
<td>3.630</td>
<td>3.950</td>
<td>4.270</td>
<td>4.590</td>
</tr>
<tr>
<td>6</td>
<td>0.130</td>
<td>0.450</td>
<td>0.770</td>
<td>1.090</td>
<td>1.410</td>
<td>1.730</td>
<td>2.050</td>
<td>2.370</td>
<td>2.690</td>
<td>3.010</td>
<td>3.330</td>
<td>3.650</td>
<td>3.970</td>
<td>4.290</td>
<td>4.610</td>
</tr>
<tr>
<td>7</td>
<td>0.150</td>
<td>0.470</td>
<td>0.790</td>
<td>1.110</td>
<td>1.430</td>
<td>1.750</td>
<td>2.070</td>
<td>2.390</td>
<td>2.710</td>
<td>3.030</td>
<td>3.350</td>
<td>3.670</td>
<td>3.990</td>
<td>4.310</td>
<td>4.630</td>
</tr>
<tr>
<td>8</td>
<td>0.170</td>
<td>0.490</td>
<td>0.810</td>
<td>1.130</td>
<td>1.450</td>
<td>1.770</td>
<td>2.090</td>
<td>2.410</td>
<td>2.730</td>
<td>3.050</td>
<td>3.370</td>
<td>3.690</td>
<td>4.010</td>
<td>4.330</td>
<td>4.650</td>
</tr>
<tr>
<td>9</td>
<td>0.190</td>
<td>0.510</td>
<td>0.830</td>
<td>1.150</td>
<td>1.470</td>
<td>1.790</td>
<td>2.110</td>
<td>2.430</td>
<td>2.750</td>
<td>3.070</td>
<td>3.390</td>
<td>3.710</td>
<td>4.030</td>
<td>4.350</td>
<td>4.670</td>
</tr>
<tr>
<td>A</td>
<td>0.210</td>
<td>0.530</td>
<td>0.850</td>
<td>1.170</td>
<td>1.490</td>
<td>1.810</td>
<td>2.130</td>
<td>2.450</td>
<td>2.770</td>
<td>3.090</td>
<td>3.410</td>
<td>3.730</td>
<td>4.050</td>
<td>4.370</td>
<td>4.690</td>
</tr>
<tr>
<td>B</td>
<td>0.230</td>
<td>0.550</td>
<td>0.870</td>
<td>1.190</td>
<td>1.510</td>
<td>1.830</td>
<td>2.150</td>
<td>2.470</td>
<td>2.790</td>
<td>3.110</td>
<td>3.430</td>
<td>3.750</td>
<td>4.070</td>
<td>4.390</td>
<td>4.710</td>
</tr>
<tr>
<td>C</td>
<td>0.250</td>
<td>0.570</td>
<td>0.890</td>
<td>1.210</td>
<td>1.530</td>
<td>1.850</td>
<td>2.170</td>
<td>2.490</td>
<td>2.810</td>
<td>3.130</td>
<td>3.450</td>
<td>3.770</td>
<td>4.090</td>
<td>4.410</td>
<td>4.730</td>
</tr>
<tr>
<td>D</td>
<td>0.270</td>
<td>0.590</td>
<td>0.910</td>
<td>1.230</td>
<td>1.550</td>
<td>1.870</td>
<td>2.190</td>
<td>2.510</td>
<td>2.830</td>
<td>3.150</td>
<td>3.470</td>
<td>3.790</td>
<td>4.110</td>
<td>4.430</td>
<td>4.750</td>
</tr>
<tr>
<td>E</td>
<td>0.290</td>
<td>0.610</td>
<td>0.930</td>
<td>1.250</td>
<td>1.570</td>
<td>1.890</td>
<td>2.210</td>
<td>2.530</td>
<td>2.850</td>
<td>3.170</td>
<td>3.490</td>
<td>3.810</td>
<td>4.130</td>
<td>4.450</td>
<td>4.770</td>
</tr>
<tr>
<td>F</td>
<td>0.310</td>
<td>0.630</td>
<td>0.950</td>
<td>1.270</td>
<td>1.590</td>
<td>1.910</td>
<td>2.230</td>
<td>2.550</td>
<td>2.870</td>
<td>3.190</td>
<td>3.510</td>
<td>3.830</td>
<td>4.150</td>
<td>4.470</td>
<td>4.790</td>
</tr>
</tbody>
</table>

NOTES: (1) Voltages in the chart represent the switch point between steps. For example, 0.010 volts is the switch point between 000 and 001.
(2) Due to quantizing error, the center of each step is the chart value minus 10 millivolts. The last step does not overflow.
(3) The first step (') equals 10 millivolts; the last step (") equals 30 millivolts.
Main Program Flow

Entry from Reset

Initialize Ports, Slack Enable Timer Interrupt

Do an A/D Conversion

Subtract 30 Hex from A/D Result

Is Result Negative?

Negate Result & Put a Minus Sign in BCDLO

Transfer Result to X-Reg.

Go to A (BCD)

Branch Subroutine Delay

Binary to BCD Subroutine Conversion

Entry from $BE

Mask Timer Int. and Clear BCDLO Reg.

X-Reg. = 0?

Decrement X-Reg. Increment BCDLO Reg.

Mask Off High Nibble of BCDLO Reg.

Is BCDLO Register $OA?

Add $06 to BCDLO Register

Clear BCDLO Register

Increment BCD Hi Register

NOTE:
When the timer interrupts the main program flow, go to the timer interrupt routine (next page). When the timer interrupt routine is finished, the MCU will automatically return to the next operation in the main program flow.

FIGURE 7 — Temperature Sensor (TEMPNS) Program Flow Chart
(Sheet 1 of 2)
PAGE 001  BOOT .SA:1  BOOT

00001  NAM BOOT
00002  OPT LLEN=100
00003
00004
00005
00006
00007  0000  A PORTA EQU 0
00008  0001  A PORTB EQU 1
00009  0005  A DDRB EQU 5
00010
00011
00012
00013  0080  ORG $80  NORMALLY THE START LOCATION WILL BE
00014A  0080  9C  RSP THE USER RESET VECTOR.
00015A  0081  2F 16  0099  BIN EXIT TEST INTERRUPT PIN FOR BOOTSTRAP ENTRY.
00016A  0083  A6 03  A LDA #503 MAKE PORTB BITS 0 AND 1 OUTPUTS.
00017A  0085  B7 01  A STA PORTB RESET 12-BIT COUNTER.
00018A  0087  B7 05  A STA DDRB
00019A  0089  13 01  A BCLR 1,PORTB ENABLE 12-BIT COUNTER.
00020A  008B  AE 40  A LOAD LDX #540
00021A  008D  B6 00  A LDA PORTA PORTA IS INPUT WHEN COMING OUT OF RESET.
00022A  008F  F7  A STA .X STORE CONTENTS OF PORTA TO RAM.
00023A  0090  11 01  A BCLR 0,PORTB INCREMENT 12-BIT COUNTER.
00024A  0092  10 01  A BSET 0,PORTB
00025A  0094  3C  INCX
00026A  0095  2A F6  008D  BPL LOAD+2 CONTINUE LOADING TILL X REG=80.
00027A  0097  BC 40  A JMP 540 JUMP TO RAM AND EXECUTE PROGRAM.
00028A  0099  CC 0100  A EXIT JMP $100 JUMP TO USER MAIN PROGRAM OR EXECUTE FROM HERE.
00029
END
TOTAL ERRORS 00000--00000
PAGE 001 SLFTST .SA:1 A/DTST

00001 NAM A/DTST

00002 OPT LLEN=100

00003A 0000 ORG $00

00004 *****LABEL DEFINITION*****

00005 0000 A PORTA EQU 0

00006 0001 A PORTB EQU 1

00007 0002 A PORTC EQU 2

00008 0003 A PORTD EQU 3

00009 0004 A DDRA EQU 4

00010 0005 A DDRB EQU 5

00011 0006 A DDCR EQU 6

00012 0007 A NC7 EQU 7

00013 0008 A TD9 EQU 8

00014 0009 A TCR EQU 9

00015 000A A MSR EQU 10

00016 000B A PCR EQU 11

00017 000C A NC2 EQU 12

00018 000D A NC3 EQU 13

00019 000E A ACR EQU 14

00020 000F A ARR EQU 15

00021 008B A LOAD EQU S8B

EXAMPLE LOCATION OF BOOTSTRAP LOADER

00022

00023 *****SEGMENT 1 DAC CALIBRATION ROUTINE*****

00024

00025A 0000 9C RSP INITIALIZE STACK

00026A 0001 AE FD A LDX #$FD INITIALIZE PORTB BUT INSURE

00027A 0003 BF 01 A STX PORTB BITS 0 AND 1 DO NOT CHANGE

00028A 0003 3F 02 A CLR PORTC

00029A 0007 AE 0F A LDX #$0F PORTC 0-3 OUTPUTS

00030A 0009 BF 06 A STX DDRC

00031A 000B AE 3F A LDX #$3F PORTB 0-5 OUTPUTS

00032A 000D BF 05 A STX DDR8

00033A 000F 1A 01 A CAL BSET 5,PORTB CLEAR DAC 12-BIT COUNTER

00034A 0011 1B 01 A BCLR 5,PORTB

00035A 0013 09 02 FD 0013 BRCLR 4,PORTC,*STOP HERE AND CAL. 0.000V IF PC=0

00036A 0016 0A 02 14 002D BRSET 5,PORTC,END1 EXIT CAL. ROUTINE IF PC3=1

00037A 0019 AE FF A LDX #$FF

00038A 001B A6 10 A LDA #$10

00039A 001D 09 01 A BCLR 4,PORTB TOGGLE DAC 12-BIT COUNTER

0003A0 001F 1B 01 A BSET 4,PORTB 16 X 255 TIMES

0003A0 004A 0021 4A DECA

0004A0 0022 26 F9 0010 BNE -*5

0004A0 0024 5A DECX

0004B0 0025 26 F4 001B BNE -*10

0004B0 0027 08 02 FD 0027 BRCLR 5,PORTC,*STOP HERE AND CAL. 5.100V IF PC5=0

0004B0 002A 09 02 E2 000F BRCLR 4,PORTC,CAL RECAL. ZERO OFFSET?

0004B0 002D BC 8B A END1 JMP LOAD RETURN TO BOOTSTRAP TO LOAD NEXT ROUTINE

0004B8.
PAGE 002 SLFTST .SA:1 A/D TEST

00050
00051
00052
00053
00054A 0040
00055

00056A 0040 1A 01 A BSET 5, PORTB RESET DAC 12-BIT COUNTER
00057A 0042 18 01 A BCLR 5, PORTB
00058A 0044 AE 08 A LDX #$08
00059A 0046 19 01 A BCLR 4, PORTB
00060A 0048 18 01 A BSET 4, PORTB
00061A 004A 5A

SECOND ROUTINE MUST START 64 BYTES LATER

00062A 004B 26 F9 0046 BNE *-5
00063A 004D AE FE A LDX #$FE
00064A 004F A6 10 A LOOP1 LDA #$10 INCREMENT DAC 12-BIT COUNTER
00065A 0051 19 01 A BCLR 4, PORTB UP TO FEB (5.090V)
00066A 0053 18 01 A BSET 4, PORTB
00067A 0055 4A

00068A 0056 26 F9 0051 BNE *-5

00069A 0058 5A

DECX

BNE LOOP1

MAKE X-REG=FF

0006A 0059 26 F4 004F BNE LOOP1

DECX

0006B 005A 5A

0006C 005C 3F 0E A ADJUST CLR ACR CLEAR CONVERSION COMPLETE FLAG
0006D 005E B6 0E A LDA ACR WAIT FOR CONVERSION (30 CYCLES)
0006E 005F 005E BPL *-2

00070A 0062 B3 0F A CPX ARR TEST RESULT FOR FF
00071A 0064 27 0A 0070 BEQ END2 AND EXIT ADJUST ROUTINE WHEN ARR=FF
00072A 0066 3C 02 A INC PORTC ELSE DECREMENT VRH VOLTAGE
00073A 0068 B3 02 A CPX PORTC

00074A 006A 26 F0 005C BNE ADJUST CONTINUE SEARCH TILL PORC=0F
00075A 006C 17 01 A FAIL1 BCLR 3, PORTB IF UNSUCCESSFUL THEN STOP
00076A 006E 20 FE 006E BRA * AFTER SETTING FLAG

00077A 0070 3A 02 A END2 DEC PORTC READJUST FOR LAST PORTC INCREMENT
00078A 0072 BC 8B A JMP LOAD RETURN TO BOOTSTRAP LOADER

00079A

299
PAGE 003 SLFTST .SA:1 A/D TEST ROUTINE****

000086

000087

000088

000089 0080 ORG $80

000090

00091A 0080 AE 01 A LDX #S01 INITIALIZE X-REG AS COUNTER

00092A 0082 1A 01 A BSET 5,PORTB RESET DAC 12-BIT COUNTER

00093A 0084 1A 01 A BCLR 5,PORTB

00094A 0086 3F 0E A CLR ACR CLEAR CONVERSION FLAG

00095A 0088 B6 0E A LDA ACR

00096A 008A 2A FC 0088 BPL *-2 WAIT FOR END OF CONVERSION

00097A 008B B3 0F A CPX ARR

00098A 008E A1 01 A CMP #S01 TEST FOR ZERO INPUT READING OF

00099A 0090 25 1A 00AC BLO FAIL2 00 OR 01 AND STOP IF ARR-REG > X-REG

00100A 0092 A6 10 A LOOP2 LDA #S10 INCREMENT DAC BY 16 STEPS

00101A 0094 19 01 A BCLR 4,PORTB TO STAY IN CENTER OF 8-BIT STEP

00102A 0096 18 01 A BSET 4,PORTB

00103A 0098 4A DECA

00104A 0099 26 F9 0094 BNE *-5

00105A 009B 3F 0E A CLR ACR CLEAR CONVERSION FLAG AND

00106A 009D B6 0E A LDA ACR WAIT FOR END OF CONVERSION

00107A 009F 2A FC 009D BPL *-2

00108A 00A1 B3 0F A CPX ARR TEST FOR CORRECT CONVERSION AND

00109A 00A3 26 07 00AC BNE FAIL2 STOP IF ARR-REG. NOT EQUAL TO X-REG.

00110A 00A5 5C INCX

00111A 00A6 26 EA 0092 BNE LOOP2 CONTINUE TESTING TILL X-REG = 00

00112A 00A8 15 01 A BCLR 2,PORTB TURN LED ON FOR TEST PASS

00113A 00AA 20 FE 00AA BRA * AND STOP

00114A 00AC 17 01 A FAIL2 BCLR 5,PORTB TURN LED ON FOR TEST FAIL

00115A 00AE 20 FE 00AE BRA * AND STOP

00116A END

TOTAL ERRORS 00000--00000

300
TEMPSN

****TEMPERATURE DISPLAY PROGRAM****

****REGISTER ADDRESS DEFINITION****

A PORTB EQU 1
A PORTC EQU 2
A DDRB EQU 5
A DDRC EQU 6
A TDR EQU 8
A TCR EQU 9
A ACR EQU 14
A ARR EQU 15
A BCDHI EQU $40
A BCDLO EQU $41
A SEVSEG EQU *
A SEGMNT EQU $42

ORG $80

****7-SEGMENT DISPLAY LOOKUP TABLE****

A SEVSEG EQU *

A FCB 00000000 0
A FCB 00001101 1
A FCB 00010000 2
A FCB 00011000 3
A FCB 00001100 4
A FCB 00000010 6
A FCB 00000000 8
A FCB 00011111 -
A FCB 00011111 BLANK

ORG $90

START RSP

LDA #5F7 INITIALIZE PORTC TO SELECT THE
STA PORTC MOST SIGNIFICANT DISPLAY FIRST
STA SEGMNT

LDA #5F7 MAKE PORTCO-2 OUTPUTS
STA DDRC

LDA #5FF INITIALIZE PORTB TO
STA PORTB TURN OFF ALL DISPLAYS

LDA #5FF PRESET TIMER FOR PRESCALE OF 128
STA TCR AND UNMASK TIMER INTERRUPT
00055A 00A7 9A       ADC    SEI
00056A 00A8 3F 06    A  CLR    ACR    CLEAR CONVERSION COMPLETE FLAG
00057A 00A9 B6 0E    A  LDA    ACR    AND SELECT CHANNEL 0
00058A 00AC 2A FC    00AA  BPL    *-2    CHECK FOR A CONVERSION COMPLETE
00059A 00AE B6 0F    A  LDA    ABR    ADJUST THE RESULT REGISTER
00060A 00B0 A2 3A    A  SBC    #$30    SO ZERO DEGREES F=30 HEX
00061A 00B2 3F 40    A  CLR    BCDHI
00062A 00B4 A1 CF    A  CMP    #$CF
00063A 00B6 25 05    00BD  BLO    CONVRT CHECK FOR NEGATIVE NUMBER
00064A 00BB 40    A  NEGA
00065A 00B9 AE 0A    A  LOX    #$50A    STORE A MINUS SIGN IN BCDHI
00066A 00BB BF 40    A  STX    BCDHI
00067A 00BD 97    CONVRT TAX    LOAD X REG WITH VALUE
00068A 00BE AD 05    00C5  BSR    BCD    AND BRANCH TO CONVERSION ROUTINE
00069A  *    NOTE: MANY OTHER TASKS CAN BE PERFORMED BEFORE
00070A  *    DOING ANOTHER CONVERSION.
00071A  *
00072A  *
00073A 00C0 9A    CLI
00074A 00C1 AD 24    00E7  BSR    DELAY    KILL TIME BEFORE DOING ANOTHER CONVERSION
00075A 00C3 20 E2    00A7  BRA    ADC    GO CHECK THE TEMPERATURE AGAIN
00076A  *
00077A  *
00078A  *
00079A  *
00080A  *    THE X REGISTER ENTERS WITH THE BINARY VALUE
00081A  *    AND EXITS WITH 00. THE CONVERTED BINARY TO
00082A  *    BCD VALUES ARE PLACED IN RAM FOR THE DISPLAY ROUTINE.
00083A  *
00084A  *
00085A 00C5 9B    A  BCD    EQU    *
00086A 00C6 3F +1    A  CLR    BCDLO    BCD LOW BYTE INITIALIZE
00087A 00C8 5D    LOOP    TSTX
00088A 00C9 27 1B    00E6  BEO    EXIT    LOOP ENDS WHEN X REG=00
00089A 00C9 27 1B    00E6  BEO    EXIT    LOOP ENDS WHEN X REG=00
00090A 00CA 5A    DECX
00091A 00CC 3C +1    A  INC    BCDLO    MOVE X CONTENTS TO LOW BYTE BCD
00092A 00CE 46 0F    A  LDA    #$50F    MASK OFF HIGH NYBBLE OF LOW BYTE BCD
00093A 00DF 85 +1    A  AND    BCDLO
00094A 00E2 A1 0A    A  CMP    #$50A    CHECK IF LOW BYTE BCD NEEDS ADJUSTING
00095A 00E4 26 F2    00C8  BNE    LOOP
00096A 00E6 B6 +1    A  LDA    BCDLO
00097A 00E8 A8 06    A  ADD    #$50F    ADJUST FOR A BCD VALUE
00098A 00EA B7 +1    A  STA    BCDLO
00099A 00EC A1 A0    A  CMP    #$5A0    CHECK IF HIGH BYTE BCD NEEDS ADJUSTING
00100A 00ED 26 E8    00C8  BNE    LOOP    GO DO IT AGAIN
00101A 00EE 3F +1    A  CLR    BCDLO    CLEAR LOW BYTE BCD
00102A 00F2 3C +1    A  INC    BCDHI    AND INCREMENT HIGH BYTE BCD
00103A 00F4 20 E2    00C8  BRA    LOOP    GO DO IT AGAIN
00104A 00FA 81    EXIT    RTS    END OF BCD ADJUST
00105A  *
****BCD TO 7-SEGMENT DISPLAY ROUTINE****

00117

* DISPLAYS ARE REFRESHED EVERY 30MSEC WHEN

00119

* A TIMER INTERRUPT OCCURS. THE MOST SIGNIFICANT DIGIT

00120

* IS DISPLAYED FIRST. THE ROUTINE IS FOR DISPLAYS WITH

00121

* COMMON ANODE LEDS.

00122

00123

A TMKINT EQU *

SEC SET THE CARRY BIT

00124A 00F2 99

00125A 00F3 A6 FF A LDA #$FF TURN 7-SEGMENT DRIVERS OFF

00126A 00F5 B7 01 A STA PORTB

00127A 00F7 36 42 A ROR SEGMT SELECT NEXT DISPLAY

00128A 00F9 25 04 00FF BCS LOBYTE CHECK CARRY BIT FOR RESTART

00129A 00FB A6 FB A LDA #$FB RESTART WITH MOST SIGNIFICANT DIGIT

00130A 00FD B7 42 A STA SEGMT

00131A 00FF B6 41 A LOBYTE LDA BCDLO

00132A 0101 02 42 04 0108 BRSSET 1,SEGMENT,HIBYTE

00133A 0104 44 LSRA SHIFT HIGH NYBBLE OF BCDLO

00134A 0105 44 LSRA TO LOW NYBBLE FOR DISPLAY

00135A 0106 44 LSRA

00136A 0107 44 LSRA

00137A 0108 04 42 06 0111 HIBYTE BRSSET 2,SEGMENT,DISPLY

00138A 010B B6 40 A LDA BCDHI

00139A 010D 26 02 0111 BNE DISPLAY BLANK IF ZERO

00140A 010F A6 06 A LDA #$06 

00141A 0111 A4 0F A DISPLY AND#$0F

00142A 0113 97 TAX

00143A 0114 EE 80 A LDX SEVSEG,X

00144A 0116 BF 01 A STX PORTB ENABLE DISPLAY DRIVERS

00145A 0118 B6 42 A LDA SEGMT

00146A 011A B7 02 A STA PORTC ENABLE DISPLAY

00147A 011C A6 10 A LDA #$10 LOAD TIMER FOR 2048 MACHINE CYCLES

00148A 011E B7 08 A STA TDR

00149A 0120 A6 0F A LDA #$0F RESET TIMER INTERRUPT FLAG

00150A 0122 B7 09 A STA TCH

00151A 0124 80 RTI

00152
REFERENCES AND ADDITIONAL READING


Motorola, Inc. MTS102 Silicon Temperature Sensor Data Sheet. Phoenix, AZ: Motorola Literature Distribution Center, 1981.


TELEPHONE DIALING TECHNIQUES USING THE MC6805

Prepared by
Robert Fischer
Downsview, Ontario, Canada

INTRODUCTION

Telephones and associated ancillary equipment providing intelligent features are fast becoming commonplace. Often, it is necessary for the microprocessor providing the intelligence to also dial a telephone number.

The M6805 Family microcomputers (MCU), with their proven hardware/software versatility, are ideal candidates for such applications. Illustrated here are two cost-effective methods of telephone dialing. Hardware and software is given for both Dual Tone Multi-Frequency (DTMF) and rotary-pulse type dialing.

DEMONSTRATION BOARD DESCRIPTION

Figure 1 shows the schematic of the demonstration board designed around a MC68705P3 single-chip MCU. This board is capable of pulse or DTMF dialing. The type of dialing is selected by switch S1. A 12-contact keyboard is used for input. While this is an extravagant use of I/O, it is acceptable for the purposes of a demonstration board.

Pulse dialing requires a direct connection to the telephone line. Interface to the line is made by a 600-ohm, 1:1 line transformer and a relay that provides on/off hook capability. An indicator light (LED #1) shows the current hook status.

After a power-on reset, the board is in an on-hook state (LED #1 off). The pressing of any key will result in an off-hook state without the digit being dialed. Subsequent key presses will result in the dialing of the corresponding digit. Pressing of the cancel button (S2) returns the board to the on-hook state.

The hardware and software to accomplish either form of dialing is readily applicable to any number of the M6805 Family.

ROTARY PULSE DIALING

From both a hardware and software viewpoint, pulse dialing is by far the simplest form of dialing to implement.

Pulse dialing requires that the telephone line circuit receive a make/break sequence at a 10-pulse-per-second (PPS) rate (see Figure 2). The dialing of the digit 3, for example, requires three make/break sequences. The 10-PPS rate requires the use of either a transistor or high speed relay for line looping. Note that if a low current reed relay is used, port B may be capable of driving the relay directly (eliminating the 2N3904 driver).

Subroutine PDIAL provides the proper timing sequences for pulse dialing. The routine is called with the digit to be dialed resident in the accumulator. Because the timing is not particularly critical, interrupts that can be quickly serviced are permissible.

DTMF DIALING

Dual tone multi-frequency tone dialing is considerably more complex in terms of ROM usage and external hardware. The M6805 MCU is required to generate two simultaneous sine waves of different frequencies. Table 1 shows the key pad digit and the frequencies of the corresponding tone pairs. Note that the tones fall into two groups:

<table>
<thead>
<tr>
<th>Group</th>
<th>Frequency (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Tones</td>
<td>697, 770, 852, 941</td>
</tr>
<tr>
<td>High Tones</td>
<td>1209, 1336, 1477, 1633</td>
</tr>
</tbody>
</table>

TABLE 1 - Keypad Digit and Frequencies for Tone Pairs

<table>
<thead>
<tr>
<th>Keypad Digit</th>
<th>DTONE Entry</th>
<th>Tone Pair (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$0</td>
<td>941 1336</td>
</tr>
<tr>
<td>1</td>
<td>$1</td>
<td>697 1209</td>
</tr>
<tr>
<td>2</td>
<td>$2</td>
<td>697 1336</td>
</tr>
<tr>
<td>3</td>
<td>$3</td>
<td>697 1477</td>
</tr>
<tr>
<td>4</td>
<td>$4</td>
<td>770 1209</td>
</tr>
<tr>
<td>5</td>
<td>$5</td>
<td>770 1336</td>
</tr>
<tr>
<td>6</td>
<td>$6</td>
<td>770 1477</td>
</tr>
<tr>
<td>7</td>
<td>$7</td>
<td>852 1209</td>
</tr>
<tr>
<td>8</td>
<td>$8</td>
<td>852 1336</td>
</tr>
<tr>
<td>9</td>
<td>$9</td>
<td>852 1477</td>
</tr>
<tr>
<td>A</td>
<td>$A</td>
<td>697 1633</td>
</tr>
<tr>
<td>B</td>
<td>$B</td>
<td>770 1633</td>
</tr>
<tr>
<td>C</td>
<td>$C</td>
<td>852 1633</td>
</tr>
<tr>
<td>D</td>
<td>$D</td>
<td>941 1633</td>
</tr>
<tr>
<td>*</td>
<td>$E</td>
<td>941 1209</td>
</tr>
<tr>
<td>#</td>
<td>$F</td>
<td>941 1477</td>
</tr>
</tbody>
</table>

Also note that if the seldom used keys A, B, C, and D are not required, it is not necessary to generate a 1633-Hz tone.

The method used to generate the tones uses a series of 3-bit look-up tables. Consider a sine wave that has been sampled
FIGURE 1 — Demonstration Board Schematic
at a constant interval, starting at the positive peak (see Figure 3). Sampling is continued until the next positive peak is encountered. There is, of course, some quantization error associated with this next found peak. If this group of samples were to be continuously cycled, a frequency error would result.

To cure this, continue sampling until the next peak is encountered and determine if the resultant frequency error falls within acceptable limits. Figure 3 is actually the output of a program written in BASIC for the EXORciser. This listing is included at the end of this application note. This program is used to design the look-up table for incorporation into the M6805 program according to the error rates acceptable in the end equipment.

This program prompts the user for the sample interval and the frequency of the tone which is to be generated. Sampling of the tone is thus automated and after a peak is encountered, the cumulative frequency error is calculated and displayed along with the sample count. If the user is satisfied with the percentage error, a table of the samples is generated. If the error is still unacceptable, the program continues sampling until the next peak is encountered. Note that the samples have all been "dc shifted".

This program was used to generate the look-up tables for all the tones given in Table 1 with a criteria of 1% maximum frequency error.

The subroutine DTONE actually operates on these tables to generate the DTMF tone pairs. The routine is entered with the DTMF digit (see Table 1) is resident in the accumulator. Note that interrupts cannot be tolerated by this routine.

The first task of this routine is to convert the digit into the table start addresses for the high and low tones. This routine requires that the tables be resident in page 0 ROM to allow use of indexed addressing with 0 offset. The structure chosen for the tables puts the high group tones in the right nibble and the low group tones in the left nibble. Because the tables are all of different lengths, the table end is marked by an entry of $F$. In defiance of Murphy's Law, the DTMF tables fit exactly into page 0 ROM.

Generation of the tones involves cycling around a loop which plucks a 3-bit low tone sample and adds it to the 3-bit high tone sample. The 4-bit sum is then output to a D/A converter. If the end of the table marker is encountered for either sample, the pointer must be reset to the table start. This loop also keeps track of the duration of the tone burst by counting loops in TIMEH and TIMEL.

Notice that every program path through the loop takes a constant time (122 microseconds). The actual sequence of program development was to first write the loop, determine the execution time, and then, with the sample interval defined, generate the tone tables.

The 4-bit D/A converter is economically implemented with standard 5% resistors (60 kilohms = 30 kilohms + 30 kilohms). Port B was used because of its slightly superior high output voltage drive. It is still necessary to supplement the high drive with pull-up resistors.

One unfortunate by-product of this tone generation technique is the production of subharmonics (and, of course, harmonics). This necessitates the use of an active bandpass filter. This filter consists of separate high pass and low pass sections. The filter response is shown in Figure 4.

The output level to the telephone line is adjusted with a 470-ohm resistor in series with the line transformer primary. This also provides the RX point, where received audio can be obtained for duplex communication.

Using the BASIC software at the end of this application note, the generation of custom tone groups is readily accomplished. Single tone generation is also possible by using the table entry TNOFF at the end of the given DTMF tables. This allows the muting of either the high or low group tones.

FIGURE 2 — Timing for Rotary Pulse Dialing

EXORciser is a registered trademark of Motorola Inc.
FIGURE 3 — Sine Wave Sampling

FIGURE 4 — Cumulative High-Pass, Low-Pass, and Line Transformer Response
PAGE 001  DEMO  .SA:  DEMO  -  MCS8705P3 SOURCE FOR DIALER DEMO

00001  NAM  DEMO
00002  TITLE  -  MCS8705P3 SOURCE FOR DIALER DEMO
00003  OPT  ABS.CES

00005
00006  **********************
00007  *
00008  *
00009  *
00010  *
00011  *
00012  *
00013  *
00014  **********************
00015
00016  *
00017  *
00018  A 0784
00019  0784  SF  A
00020  FCB  $01011111

00021
00022  *
00023  *
00024  PORT DEFINITION

00025  0000  A  PORTA EDU $0  PORT A I/O
00026  0020  A  PORTB EDU $1  PORT B I/O
00027  0040  A  PORTC EDU $2  PORT C I/O
00028  0060  A  PORTD EDU $4  PORT A DDR
00029  0080  A  PORTED EDU $5  PORT B DDR
00030  0100  A  PORTFD EDU $6  PORT C DDR
00031  0120  A  TABDATA EDU $8  TIMER DATA REGISTER
00032  0140  A  TMACON EDU $9  TIMER CONTROL REGISTER

0034
0035  *
0036  RAM ALLOCATION

00037  0000  ORG $40
00038  0040  0001  A  TIMETO RMB 1  TONE TIME (MS)
00039  0040  0001  A  TIMETL RMB 1  TONE TIME (LS)
00040  0040  0001  A  STARTL RMB 1  HI TONE TABLE START
00041  0043  0001  A  STARTL RMB 1  LO TONE TABLE START
00042  0043  0001  A  TONEY RMB 1  HI TONE CURRENT POINTER
00043  0045  0001  A  TONEY RMB 1  LO TONE CURRENT POINTER
00044  0046  0001  A  SCATCH RMB 1  TEMPORARY SCRATCH
00045  0047  0001  A  LSTKEY RMB 1  LAST DIALED DIGIT USED
ORG $100

* COLD START.
* INITIALIZE PORTS.

* CHECK KEYBOARD FOR CLOSURE

* SCAN JSR KEYIN

* NO CLOSURE- CHECK "ON HOOK" SWITCH

* DIGIT HAS BEEN PUSHED.
* IF PREVIOUSLY ON-HOOK, COME OFF-HOOK
* AND WAIT FOR NEXT KEY.

* SEE IF PULSE DIALING OR DTMF

* DIAL BRCLR 0,PORTC,PULSE

* "+" AND "#" KEYS ARE NOT RECOGNIZED

* PULSE DIALING

* A PULSE CMP #A

* BHI SCAN

* JSR PDIAL

* BRA SCAN
PAGE 003 DEMO .BA+1 DEMO - MC6805P3 SOURCE FOR DIALER DEMO

00094
00095
00096
00097
00098

*****************************************************************************

00099
00100
00101
00102
00103
00104

*****************************************************************************

00105
00106
00107
00108
00109
00110

*****************************************************************************

00111
00112
00113
00114
00115
00116
00117
00118
00119
00120

*****************************************************************************

00121
00122
00123
00124
00125
00126
00127
00128
00129
00130

*****************************************************************************

00131
00132
00133
00134
00135
00136
00137
00138
00139
00140

*****************************************************************************

00141
00142
00143
00144
00145
00146
00147

*****************************************************************************

00148

*****************************************************************************

00149

*****************************************************************************

00150

*****************************************************************************

00151

*****************************************************************************

00152

*****************************************************************************

00153

*****************************************************************************

00154

*****************************************************************************

00155

*****************************************************************************

00156

*****************************************************************************

00157

*****************************************************************************

00158

*****************************************************************************

00159

*****************************************************************************

00160

*****************************************************************************

00161

*****************************************************************************

00162

*****************************************************************************

00163

*****************************************************************************

00164

*****************************************************************************

00165

*****************************************************************************

00166

*****************************************************************************

00167

*****************************************************************************

00168

*****************************************************************************

00169

*****************************************************************************

00170

*****************************************************************************

00171

*****************************************************************************

00172

*****************************************************************************

00173

*****************************************************************************

00174

*****************************************************************************

00175

*****************************************************************************

00176

*****************************************************************************

00177

*****************************************************************************

00178

*****************************************************************************

00179

*****************************************************************************

00180

*****************************************************************************

00181

*****************************************************************************

00182

*****************************************************************************

00183

*****************************************************************************

00184

*****************************************************************************

00185

*****************************************************************************

00186

*****************************************************************************

00187

*****************************************************************************

00188

*****************************************************************************

00189

*****************************************************************************

00190

*****************************************************************************

00191

*****************************************************************************

00192

*****************************************************************************

00193

*****************************************************************************

00194

*****************************************************************************

00195

*****************************************************************************

00196

*****************************************************************************

00197

*****************************************************************************

00198

*****************************************************************************

00199

*****************************************************************************

00200
PAGE 004  DEMO  .SA:1  DEMO  - MC6870525 SOURCE FOR DIALER DEMO

00:48A 015D 06 19  A   LDA  #$25
00:49A 01:5F 20  D1  0132  BRA  WAITMS
00:50A  *  
00:51A 01:61 9D  NCARRY  LDP  NO CARRY- BURN 6 USECS
00:52A 01:62 20  F7  015E  BRA  CONT
00:53A  *  
00:54A  *  GET  LO  TONE
00:55A  *  
00:56A 01:64  BE  45  A  GETLO  LDX  TONEL  LO  TONE  CURRENT  POINTER
00:57A 01:66  F6  LDA  #X  GET  BYTE
00:58A  *  LO  TONES  ARE  IN  LEFT  NIBBLE
00:59A  *  SEE  IF  TABLE  END  (MS  BIT  =1)
00:5A A 01:67 28  64  016D  BMI  GETNLO
00:5B A 01:65 9D  NOP
00:5C A 01:6A 9D  NOP  FILL  FOR  CONSTANT  TIME
00:5D A 01:68 20  03  0170  BRA  SHFL0
00:5E A  
00:5F A  *  TABLE  END-  RESET  TO  TABLE  START
00:60A  
00:61A 01:6D  BE  43  A  GETNLO  LDX  STARTL
00:62A 01:6F  F6  LDA  #X
00:63A  
00:64A 01:70 44  S-FLO  LSRA
00:65A 01:71 44  LSRA
00:66A 01:72 44  LSRA
00:67A 01:73 44  LSRA
00:68A  *  LO  BYTE  VALID-  SAVE
00:69A 01:74  B7  46  A  STA  SCRTCH-
00:6A A 01:75 5C  *  FIX  POINTER  AND  SAVE
00:6B A 01:76 5C  INX
00:6C A 01:77  BF  45  A  STX  TONEL
00:6D A  
00:6E A  *  GET  HI  TONE
00:6F A  
00:70A 01:72  BE  44  A  LDX  TONEH  HI  TONE  CURRENT  POINTER
00:71A 01:76  F5  LDP  #X  GET  BYTE
00:72A  *  SEE  IF  TABLE  END
00:73A 01:7C 44  0F  A  AND  #$F
00:74A 01:7E  44  3F  A  CMP  #$F
00:75A 01:80 27  04  0186  BCD  GENHI
00:76A 01:82  9D  NOP
00:77A 01:83  9D  NOP  FILL  FOR  CONSTANT  TIME
00:78A 01:84 20  03  0189  BRA  ADDNUM
00:79A  
00:7A A  *  -1  TABLE  END-  RESET  TO  START
00:7B A  
00:7C A 01:86  BE  42  A  GENHI  LDX  STARTH
00:7D A 01:88  F6  LDA  #X
00:7E A  *  GET  HI  NIBBLE-  FIX  X
00:7F A 01:89  5C  ADDNUM  INX
00:80A 01:8A  BF  44  A  STX  TONEH
00:81A  *  
00:82A  
00:83A  *  ADD  LO  AND  HI  TONES
00:84A  
00:85A 01:8C  BB  46  A  ADD  SCRTCH
00:86A  *  OUTPLT  THIS  SAMPLE

312
PAGE 305 DEMO .BAS DEMO - YC58705P3 SOURCE FOR DIALER DEMO

00236A 0.2E B7 01 A STA PORTB
00237A 05 A5 H8
00238A + KEEP GOING
00239A 05 A5 H8

FILL FOR 122 USEC LOOP

0024E6
0024F6
0024FE

+ TONE PAIR LOOK-UP TABLE
00257A

0025EA 21B5 9C A TTAB FCB TN1336 (2)
0025FA 0.96 93 A FCB TN941
00260A 0.97 80 A FCB TN1209 (1)
00261A 0.98 80 A FCB TN637
00262A 0.99 90 A FCB TN1235 (2)
00263A 0.9A 90 A FCB TN637
00264A 0.9B 90 A FCB TN1229 (4)
00265A 0.9C 90 A FCB TN1336 (5)
00266A 0.9D 90 A FCB TN770
00267A 0.9E 90 A FCB TN477 (6)
00268A 0.9F 90 A FCB TN770
00269A 0.9A 90 A FCB TN1209 (7)
0026EA 0.9B 90 A FCB TN852
0026FA 0.9C 90 A FCB TN1336 (8)
00270A 0.9D 90 A FCB TN852
00271A 0.9E 90 A FCB TN1477 (9)
00272A 0.9F 90 A FCB TN852
00273A 0.9A 90 A FCB TN1633 (A)
00274A 0.9B 90 A FCB TN637
00275A 0.9C 90 A FCB TN1633 (B)
00276A 0.9D 90 A FCB TN770
00277A 0.9E 90 A FCB TN1633 (C)
00278A 0.9F 90 A FCB TN852
00279A 0.9A 90 A FCB TN1633 (D)
0027EA 0.9B 90 A FCB TN941
0027FA 0.9C 90 A FCB TN1209 (E)
00280A 0.9D 90 A FCB TN941
00281A 0.9E 90 A FCB TN941

00285.
00286A

* KEYIN
00287A

* SCANS KEYBOARD FOR PRESSED KEY.
00288A

* WHEN A NEW KEY IS PRESSED, RETURN
00289A

* WITH CARRY SET AND KEY VALUE IN ACC.
0028EA

0028FA 21B5 AD IF 01D6 KEYIN 38R CHECK CHECK FOR INPUT
0028EA 01B7 25 15 01DE BCS KEYIN ANY KEY?
PAGE 007 DEMO DEMO - YC68705SP3 SOURCE FOR DIALER DEMO

0336A 01F4 81 RTS

00318
00319
00320
00321
00322
00323A 01F5 4D * DIAL
00324A 01FE 26 02 01FD BNE PDIAL1
00325A 01FE 96 0A A LDA #$A
00327A 01FA B7 46 A PDIAL1 STA SCRATCH SAVE
00328A * MAKE A PULSE
00329A 01FC :5 02 A PDIAL2 BCLR 2,PORTC ON HOOK
00332A 01FE 16 3D A LDA #$1
00333A 0200 CD 0132 A JSR WAITMS
00332A 0203 14 02 A BSET 2,PORTC OFF HOOK
00333A 0205 A6 27 A LDA =$9
00334A 0207 CD 0132 A JSR WAITMS
00335A 020A 3A 46 A DEC SCRATCH
00336A 020C 2E SE 01FC BNE PDIAL2
00337A * WAIT 200 MSEC INTER-DIGIT
00338A 020E A6 C8 A LDA #$200
00338A 0210 CC 0132 A JMP WAITMS
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Type</th>
<th>ORG</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>00345A</td>
<td>0080</td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>00346A</td>
<td>0080</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>00347A</td>
<td>0081</td>
<td></td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>00348A</td>
<td>0082</td>
<td></td>
<td></td>
<td>03</td>
</tr>
<tr>
<td>00349A</td>
<td>0083</td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>00350A</td>
<td>0084</td>
<td></td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>00351A</td>
<td>0085</td>
<td></td>
<td></td>
<td>03</td>
</tr>
<tr>
<td>00352A</td>
<td>0086</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>00353A</td>
<td>0087</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>00354A</td>
<td>0088</td>
<td></td>
<td></td>
<td>42</td>
</tr>
<tr>
<td>00355A</td>
<td>0089</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>00356A</td>
<td>008A</td>
<td></td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>00357A</td>
<td>008B</td>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>00358A</td>
<td>008C</td>
<td></td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>00359A</td>
<td>008D</td>
<td></td>
<td></td>
<td>57</td>
</tr>
<tr>
<td>0035A0</td>
<td>008E</td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>0035B0</td>
<td>008F</td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>0035C0</td>
<td>0090</td>
<td></td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>0035D0</td>
<td>0091</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>0035E0</td>
<td>0092</td>
<td></td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>0035F0</td>
<td>0093</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>003600</td>
<td>0094</td>
<td></td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>003610</td>
<td>0095</td>
<td></td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>003620</td>
<td>0096</td>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>003630</td>
<td>0097</td>
<td></td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>003640</td>
<td>0098</td>
<td></td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>003650</td>
<td>0099</td>
<td></td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>003660</td>
<td>009A</td>
<td></td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>003670</td>
<td>009B</td>
<td></td>
<td></td>
<td>3F</td>
</tr>
<tr>
<td>003680</td>
<td>009C</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>003690</td>
<td>009D</td>
<td></td>
<td></td>
<td>05</td>
</tr>
<tr>
<td>0036A0</td>
<td>009E</td>
<td></td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>0036B0</td>
<td>009F</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>0036C0</td>
<td>00A0</td>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>0036D0</td>
<td>00A1</td>
<td></td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>0036E0</td>
<td>00A2</td>
<td></td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>0036F0</td>
<td>00A3</td>
<td></td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>003700</td>
<td>00A4</td>
<td></td>
<td></td>
<td>72</td>
</tr>
<tr>
<td>003710</td>
<td>00A5</td>
<td></td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>003720</td>
<td>00A6</td>
<td></td>
<td></td>
<td>51</td>
</tr>
<tr>
<td>003730</td>
<td>00A7</td>
<td></td>
<td></td>
<td>34</td>
</tr>
<tr>
<td>003740</td>
<td>00A8</td>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>003750</td>
<td>00A9</td>
<td></td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>003760</td>
<td>00AA</td>
<td></td>
<td></td>
<td>03</td>
</tr>
<tr>
<td>003770</td>
<td>00AB</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>003780</td>
<td>00AC</td>
<td></td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>003790</td>
<td>00AD</td>
<td></td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>0037A0</td>
<td>00AE</td>
<td></td>
<td></td>
<td>77</td>
</tr>
<tr>
<td>0037B0</td>
<td>00AF</td>
<td></td>
<td></td>
<td>76</td>
</tr>
<tr>
<td>0037C0</td>
<td>00B0</td>
<td></td>
<td></td>
<td>63</td>
</tr>
<tr>
<td>0037D0</td>
<td>00B1</td>
<td></td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>0037E0</td>
<td>00B2</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>PAGE 009_demo</td>
<td>.SA1 demos</td>
<td>- MC68705P3 source for dialer demo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>-----------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00037A 00B3</td>
<td>13</td>
<td>A FCB $1.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00038A 00B4</td>
<td>06</td>
<td>A FCB $0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00039A 00B5</td>
<td>17</td>
<td>A FCB $1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040A 00B6</td>
<td>24</td>
<td>A FCB $2.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00041A 00B7</td>
<td>41</td>
<td>A FCB $4.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00042A 00B8</td>
<td>60</td>
<td>A FCB $6.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00043A 00B9</td>
<td>73</td>
<td>A FCB $7.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00044A 00B0</td>
<td>76</td>
<td>A FCB $7.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00045A 00B1</td>
<td>57</td>
<td>A FCB $5.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00046A 00B2</td>
<td>34</td>
<td>A FCB $3.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00047A 00B3</td>
<td>11</td>
<td>A FCB $1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00048A 00B4</td>
<td>00</td>
<td>A FCB $0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00049A 00B5</td>
<td>03</td>
<td>A FCB $0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00050A 00B6</td>
<td>16</td>
<td>A FCB $1.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00051A 00B7</td>
<td>3F</td>
<td>A FCB $3.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00052A 00B8</td>
<td>57</td>
<td>A TX1477 FCB $5.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00053A 00B9</td>
<td>65</td>
<td>A FCB $6.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00054A 00B0</td>
<td>F1</td>
<td>A FCB $F.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00055A 00B1</td>
<td>70</td>
<td>A TX852 FCB $7.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00056A 00B2</td>
<td>63</td>
<td>A FCB $6.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00057A 00B3</td>
<td>46</td>
<td>A FCB $4.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00058A 00B4</td>
<td>27</td>
<td>A FCB $2.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00059A 00B5</td>
<td>03</td>
<td>A FCB $0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00060A 00B6</td>
<td>00</td>
<td>A FCB $0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00061A 00B7</td>
<td>11</td>
<td>A FCB $1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00062A 00B8</td>
<td>35</td>
<td>A FCB $3.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00063A 00B9</td>
<td>57</td>
<td>A FCB $5.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00064A 00C0</td>
<td>75</td>
<td>A FCB $7.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00065A 00C1</td>
<td>72</td>
<td>A FCB $7.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00066A 00C2</td>
<td>60</td>
<td>A FCB $6.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00067A 00C3</td>
<td>42</td>
<td>A FCB $4.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00068A 00C4</td>
<td>16</td>
<td>A FCB $1.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00069A 00C5</td>
<td>07</td>
<td>A FCB $0.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00070A 00C6</td>
<td>04</td>
<td>A FCB $0.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00071A 00C7</td>
<td>20</td>
<td>A FCB $2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00072A 00C8</td>
<td>41</td>
<td>A FCB $4.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00073A 00C9</td>
<td>64</td>
<td>A FCB $6.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00074A 00DA</td>
<td>75</td>
<td>A FCB $7.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00075A 00DB</td>
<td>55</td>
<td>A FCB $5.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00076A 00DC</td>
<td>31</td>
<td>A FCB $3.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00077A 00DD</td>
<td>11</td>
<td>A FCB $1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00078A 00DE</td>
<td>05</td>
<td>A FCB $0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00079A 00DF</td>
<td>17</td>
<td>A FCB $1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00080A 00E0</td>
<td>25</td>
<td>A FCB $2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00081A 00E1</td>
<td>51</td>
<td>A FCB $5.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00082A 00E2</td>
<td>61</td>
<td>A FCB $6.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00083A 00E3</td>
<td>F4</td>
<td>A FCB $F.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00084A 00E4</td>
<td>77</td>
<td>A TX1633 FCB $7.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00085A 00E5</td>
<td>55</td>
<td>A FCB $5.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00086A 00E6</td>
<td>31</td>
<td>A FCB $3.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00087A 00E7</td>
<td>11</td>
<td>A FCB $1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00088A 00E8</td>
<td>05</td>
<td>A FCB $0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00089A 00E9</td>
<td>17</td>
<td>A FCB $1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00090A 00EA</td>
<td>25</td>
<td>A FCB $2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00091A 00EB</td>
<td>51</td>
<td>A FCB $5.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00092A 00EC</td>
<td>61</td>
<td>A FCB $6.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00093A 00ED</td>
<td>F4</td>
<td>A FCB $F.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00094A 00EE</td>
<td>77</td>
<td>A TX941 FCB $7.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00095A 00EF</td>
<td>55</td>
<td>A FCB $5.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00096A 00EG</td>
<td>31</td>
<td>A FCB $3.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00097A 00EH</td>
<td>11</td>
<td>A FCB $1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00098A 00EI</td>
<td>05</td>
<td>A FCB $0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00099A 00EJ</td>
<td>17</td>
<td>A FCB $1.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100A 00EK</td>
<td>25</td>
<td>A FCB $2.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101A 00EL</td>
<td>51</td>
<td>A FCB $5.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00102A 00EM</td>
<td>61</td>
<td>A FCB $6.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00103A 00EN</td>
<td>F4</td>
<td>A FCB $F.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Value</td>
<td>Type</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>00455A</td>
<td>00ED</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00456A</td>
<td>00EE</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00457A</td>
<td>00EF</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00458A</td>
<td>00F0</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00459A</td>
<td>00F1</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00460A</td>
<td>00F2</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00461A</td>
<td>00F3</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00462A</td>
<td>00F4</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00463A</td>
<td>00F5</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00464A</td>
<td>00F6</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00465A</td>
<td>00F7</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00466A</td>
<td>00F8</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00467A</td>
<td>00F9</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00468A</td>
<td>00FA</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00469A</td>
<td>00FB</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00470A</td>
<td>00FC</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00471A</td>
<td>00FD</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
<tr>
<td>00472A</td>
<td>00FE</td>
<td>A</td>
<td>TNOFF</td>
<td></td>
</tr>
<tr>
<td>00473A</td>
<td>00FF</td>
<td>A</td>
<td>FCB</td>
<td></td>
</tr>
</tbody>
</table>

00475  0080  A TN1209 EQU  TN697  (SHARE START ADDRESS)
00479
00479
00480
00481A 07F8
00482A 07F8 0100 A ORG $7F8
00483A 07FA 0100 A FDB START TIMER INTERRUPT
00484A 07FC 0100 A FDB START HARDWARE INTERRUPT
00485A 07FE 0100 A FDB START SWI
00486
TOTAL ERRORS 00000--00000

0:08 ADDNUM 00192 00139*
0:06 CHECK 00237 00269 00292*
0:18 CONT 00144+00152
0:1A DIAL 00066 00076*
0:11 DODIAL 00075 00082*
0:13 DTCNE 00083 00021*
0:18 JENVH 00189 00196*
0:14 GTCLO 00144 00156*
0:16 GTNCLO 00160 00167*
0:03 JOTKEY 00235 00300 00302 00304 00306 00315*
0:09 KEY1 00294+00298
0:0E KEY1: 00299 00379 00280*
0:1C KEY22 00265 00274+00281
0:15 KEYIN 00065 00257*
0:07 LSTKEY 00455+00263 00275 00280 00282
0:16 MARRY 00142 00151*
0:15 PIAL 00291 00323*
0:1A PIAL1 00324 00327*
0:1C PIAL2 00329+00336
0:20 PORTA 00025+00055 00293
0:04 PORTAD 00028*
0:02 PORTB 00026+00056 00206 00300 00302 00304 00305
0:05 PORTED 00029+00059
0:02 PORTC 00027+00057 00507 00059 00076 00077 00082 00329 00332
0:06 PORTCD 00030+00061*
0:19 PULSE 00032 00089*
0:0E SCA 00065+00066 00027 00064 00039 00092
0:0E SCTR 00044+00177 00204 00327 00335
0:05 SHFO 00163 00172*
0:00 START 00055+00482 00463 00484 00485
0:02 STARTI 00040+00125 00196
0:03 STARTL 00044+00128 00167
0:04 TIMEM 00038+00134 00143
0:04 TIPEL 00039+00136 00141
0:29 VACON 00032*
0:06 VDRAI 00071*
0:20 TN1222 00222 00226 00232 00246 00475*
0:23 TN1234 00218 00222 00228 00234 00374*
0:04 TW1477 00224 00230 00236 00248 00412*
0:0D9 TN1633 00238 00240 00242 00244 00435*
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00221</td>
</tr>
<tr>
<td>0001</td>
<td>00223</td>
</tr>
<tr>
<td>0002</td>
<td>00225</td>
</tr>
<tr>
<td>0003</td>
<td>00239</td>
</tr>
<tr>
<td>0004</td>
<td>00346</td>
</tr>
<tr>
<td>0005</td>
<td>00475</td>
</tr>
<tr>
<td>0006</td>
<td>00227</td>
</tr>
<tr>
<td>0007</td>
<td>00229</td>
</tr>
<tr>
<td>0008</td>
<td>00231</td>
</tr>
<tr>
<td>0009</td>
<td>00241</td>
</tr>
<tr>
<td>000A</td>
<td>00382</td>
</tr>
<tr>
<td>000B</td>
<td>00233</td>
</tr>
<tr>
<td>000C</td>
<td>00235</td>
</tr>
<tr>
<td>000D</td>
<td>00237</td>
</tr>
<tr>
<td>000E</td>
<td>00243</td>
</tr>
<tr>
<td>000F</td>
<td>00415</td>
</tr>
<tr>
<td>0010</td>
<td>00219</td>
</tr>
<tr>
<td>0011</td>
<td>00245</td>
</tr>
<tr>
<td>0012</td>
<td>00247</td>
</tr>
<tr>
<td>0013</td>
<td>00249</td>
</tr>
<tr>
<td>0014</td>
<td>00445</td>
</tr>
<tr>
<td>0015</td>
<td>00432</td>
</tr>
<tr>
<td>0016</td>
<td>00126</td>
</tr>
<tr>
<td>0017</td>
<td>00184</td>
</tr>
<tr>
<td>0018</td>
<td>00200</td>
</tr>
<tr>
<td>0019</td>
<td>00201</td>
</tr>
<tr>
<td>001A</td>
<td>00420</td>
</tr>
<tr>
<td>001B</td>
<td>00129</td>
</tr>
<tr>
<td>001C</td>
<td>00156</td>
</tr>
<tr>
<td>001D</td>
<td>00180</td>
</tr>
<tr>
<td>001E</td>
<td>00414</td>
</tr>
<tr>
<td>001F</td>
<td>00213</td>
</tr>
<tr>
<td>0020</td>
<td>00124</td>
</tr>
<tr>
<td>0021</td>
<td>00127</td>
</tr>
<tr>
<td>0022</td>
<td>00219</td>
</tr>
<tr>
<td>0023</td>
<td>00105</td>
</tr>
<tr>
<td>0024</td>
<td>00107</td>
</tr>
<tr>
<td>0025</td>
<td>00105</td>
</tr>
<tr>
<td>0026</td>
<td>00109</td>
</tr>
<tr>
<td>0027</td>
<td>00149</td>
</tr>
<tr>
<td>0028</td>
<td>00268</td>
</tr>
<tr>
<td>0029</td>
<td>00331</td>
</tr>
<tr>
<td>002A</td>
<td>00334</td>
</tr>
<tr>
<td>002B</td>
<td>00339</td>
</tr>
</tbody>
</table>
PAGE 00: TONCAL .SA:1

0100 DIM T(100)
0101 H=1
0102 P=122E-6
0103 C=0
0110 W=8+1E6
0130 PRINT "PERIOD = ";K;"USECS (Y/N)"
0120 INPUT A#
0127 IF A#="Y" THEN GOTO 120
0124 IF A#="N" THEN GOTO 121
0125 INPUT "PERIOD (USECS)=", P
0126 P=P+1E6
0130 PRINT "TONE FREQ=";
0140 INPUT F
0150 X=2*3.141593*F*P
0160 FOR N=1 TO 100
0170 T(N)=INT(3.5*COS(N*X)+4)
0171 GOSUB 8000
0190 IF T(N)<7 THEN GOTO 370
0200 RET GOTO 100
0220 Q=N*P
0230 E=(C/F-Q)*100/G
0240 GOSUB 7000
0250 PRINT "ERROR =";E;"%"
0270 PRINT "SAMPLE COUNT =", N
0275 PRINT "FREQUENCY =",F;
0280 PRINT "CONTINUE (Y/N)"
0290 INPUT A#
0300 IF A#="Y" THEN GOTO 370
0310 IF A#="N" THEN GOTO 280
0320 REM DONE - PRINT SAMPLE LIST
0330 GOSUB 9000
0350 GOTO 100
0370 NEXT N
0380 END
7000 REM PLOT SUBROUTINE
7001 PRINT CHR$(27);"a"
7002 FOR I=1 TO 50
7003 NEXT I
7005 PRINT
7006 PRINT "":";
7010 FOR I=1 TO 8
7020 =8-I
7030 PRINT I;":"
7040 REM PRINT THIS LINE
7050 FOR J=1 TO N
7060 IF "J"=L THEN GOTO 7090
7070 PRINT "":";
7080 GOTO 7100
7090 PRINT "":";
7100 NEXT J
7110 PRINT
7120 NEXT I
7120 PRINT "":";
7140 FOR I=1 TO 70
7150 PRINT "":";
7160 NEXT I
7160 W=P*1E6
PAGE 002 "TONEAL .SA:1

7170 PRINT " HORIZONTAL = "W:1" USEC"
7180 PRINT
7190 PRINT
7200 RETURN
0000 REM CREST COUNT ROUTINE- LOOK FOR A ZERO CROSSING
0010 IF T(N) > 3 THEN GOTO 0040
0020 H=0
0030 RETURN
0040 IF H=0 THEN C=C+1
0050 H=1
0060 RETURN
9000 REM PRINTER OUTPUT" SUBROUTINE
9010 FOR J=1 TO 4
9020 PRINT "2"
9030 NEXT J
9040 FOR I=1 TO 8
9050 L=E-I
9060 PRINT "2",I:" "
9070 FOR J=1 TO N
9080 IF T(J)=L THEN GOTO 910
9090 PRINT "2"," "
9100 GOTO 9120
9110 PRINT "2"," #"
9120 NEXT J
9130 PRINT "2"
9140 NEXT I
9150 PRINT "2"," "
9160 FOR I=1 TO 70
9170 PRINT "2"," I"
9180 NEXT I
9190 W=P*1E6
9200 DIGITS= 3
9210 PRINT "2"
9220 PRINT "2"," "
9230 PRINT "2"," FREQUENCY = "F1" Hz"
9240 PRINT "2"," ERROR = "E1" %"
9250 PRINT "2"," NO. SAMPLES = " I:
9260 PRINT "2"," NO. CYCLES = " IC
9270 PRINT "2"
9280 DIGITS= 0
9290 PRINT "2"
9300 FOR J=1 TO N STEP 2
9310 PRINT "2"," SAMPLE "I": = "I"(J)
9320 I=I+1
9330 IF I=1 THEN GOTO 9340
9340 PRINT "2"," SAMPLE "2": = "2"(J)
9350 PRINT "2"
9360 NEXT J
9370 PRINT "2",CHR$(12)
9380 RETURN

322
MC68HC11 Floating-Point Package

INTRODUCTION

The MC68HC11 is a very powerful and capable single-chip microcomputer. Its concise instruction set combined with six powerful addressing modes, true bit manipulation, 16-bit arithmetic operations and a second 16-bit index register make it ideal for control applications requiring both high-speed I/O and high-speed calculations.

While most applications can be implemented by using the 16-bit integer precision of the MC68HC11, certain applications or algorithms may be difficult or impossible to implement without floating-point math. The goal in writing the MC68HC11 floating-point package was to provide a fast, flexible way to do floating-point math for just such applications.

The HC11 floating-point package (HC11FP) implements more than just the four basic math functions (add, subtract, multiply, and divide); it also provides routines to convert from ASCII to floating point and from floating point to ASCII. For those applications that require it, the three basic trig functions SINe, COSine, and TANgent are provided along with some trig utility functions for converting to and from radians and degrees. The square root function is also included.

For those applications that can benefit by using both integer and floating-point operations, there are routines to convert to and from integer and floating-point format.

The entire floating-point package requires just a little over 2k bytes of memory and only requires ten bytes of page-zero RAM in addition to stack RAM. All temporary variables needed by the floating-point routines, reside on the stack. This feature makes the routines completely reentrant as long as the ten bytes of page zero RAM are saved before using any of the routines. This will allow both interrupt routines and main line programs to use the floating-point package without interfering with one another.

FLOATING-POINT FORMAT

FLOATING-POINT ACCUMULATOR FORMAT

The ten bytes of page-zero RAM are used for two software floating-point accumulators FPACC1 and FPACC2. Each five-byte accumulator consists of a one-byte exponent, a three-byte mantissa, and one byte that is used to indicate the mantissa sign.

The exponent byte is used to indicate the position of the binary point and is biased by decimal 128 ($80) to make floating-point comparisons easier. This one-byte exponent gives a dynamic range of about $1 \times 10^{\pm 38}$.

The mantissa consists of three bytes (24 bits) and is used to hold both the integer and fractional portion of the floating-point number. The mantissa is always assumed to be "normalized" (i.e., most-significant bit of the most-significant byte a one). A 24-bit mantissa will provide slightly more than seven decimal digits of precision.

A separate byte is used to indicate the sign of the mantissa rather than keeping it in twos complement form so that unsigned arithmetic operations may be used when manipulating the mantissa. A positive mantissa is indicated by this byte being equal to zero ($00$). A negative mantissa is indicated by this byte being equal to minus one ($FF$). The following table shows the error codes and their meanings are explained below.

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>No error</td>
</tr>
<tr>
<td>00</td>
<td>FF</td>
<td>Error code 1</td>
</tr>
<tr>
<td>00</td>
<td>E0</td>
<td>Error code 2</td>
</tr>
<tr>
<td>00</td>
<td>F0</td>
<td>Error code 3</td>
</tr>
<tr>
<td>00</td>
<td>FF</td>
<td>Error code 4</td>
</tr>
</tbody>
</table>
```

MEMORY FORMAT

The way that floating-point numbers are stored in memory or the "memory format" of a floating-point number is slightly different than its floating-point accumulator format. In order to save memory, floating-point numbers are stored in memory in a format called "hidden bit normalized form".

In this format, the number is stored into four consecutive bytes with the exponent residing at the lowest address. The mantissa is stored in the next three consecutive bytes with the most-significant byte stored in the lowest address. Since the most-significant bit of the mantissa in a normalized floating-point number is always a one, this bit can be used to store the sign of the mantissa. This results in positive numbers having the most-significant bit of the mantissa cleared (zero) and negative numbers having their most-significant bit set (one). An example follows:

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>490FDB</td>
<td>+3.1415927</td>
</tr>
<tr>
<td>00</td>
<td>C90FDB</td>
<td>-3.1415927</td>
</tr>
</tbody>
</table>
```

There are four routines that can be used to save and load the floating-point accumulators and at the same time convert between the floating-point accumulator and memory format. These routines are discussed in detail in FLOATING-POINT ROUTINES.

ERRORS

There are seven error conditions that may be returned by the HC11 floating-point package. When an error occurs, the condition is indicated to the calling program by setting the carry bit in the condition code register and returning an error code in the A-accumulator. The error codes and their meanings are explained below.
FLOATING-POINT ROUTINES

The following paragraphs provide a description of each routine in the floating-point package. The information provided includes the subroutine name, operation performed, subroutine size, stack space required, other subroutines that are called, input, output, and possible error conditions.

The Stack Space required by the subroutine includes not only that required for the particular routines local variables, but also stack space that is used by any other subroutines that are called including return addresses. Note that the trig functions require a good deal of stack space.

Since some applications may not require all the routines provided in the floating-point package, the description of each routine includes the names of other subroutines that it calls. This makes it easy to determine exactly which subroutines are required for a particular function.

ASCII-TO-FLOATING-POINT CONVERSION

Subroutine Name: ASCFLT
Operation: ASCII (X) → FPACC1
Size: 352 Bytes (includes NUMERIC subroutine)
Stack Space: 14 Bytes
Calls: NUMERIC, FPNORM, FLTMUL, PSHFPAC2, PULFPAC2
Input: X register points to ASCII string to convert.
Output: FPACC1 contains the floating-point number.
Error Conditions: Floating-point format error may be returned.
Notes: This routine converts an ASCII floating-point number to the format required by all of the floating-point routines. Conversion stops either when a non-decimal character is encountered before the exponent or after one or two exponent digits have been converted. The input format is very flexible. Some examples are shown below.

20.095
0.125
7.2984E+10
167.824E5
005.9357E-7
500

FLOATING-POINT MULTIPLY

Subroutine Name: FLTMUL
Operation: FPACC1 x FPACC2 → FPACC1
Size: 169 Bytes
Stack Space: 10 Bytes
Calls: PSHFPAC2, PULFPAC2, CHK0
Input: FPACC1 and FPACC2 contain the numbers to be multiplied.
Output: FPACC1 contains the product of the two floating-point accumulators. FPACC2 remains unchanged.
Error Conditions: Overflow, Underflow.
**FLOATING-POINT ADD**

Subroutine Name: FLTADD  
Operation: FPACC1 + FPACC2 → FPACC1  
Size: 194 Bytes  
Stack Space: 6 Bytes  
Calls: PSHFPAC2, PULFPAC2, CHK0  
Input: FPACC1 and FPACC2 contain the numbers to be added.  
Output: FPACC1 contains the sum of the two numbers. FPACC2 remains unchanged.  
Error Conditions: Overflow, Underflow.  
Notes: The floating-point add routine performs full signed addition. Both floating-point accumulators may have mantissas with the same or different sign.

**FLOATING-POINT SUBTRACT**

Subroutine Name: FLTSUB  
Operation: FPACC1 - FPACC2 → FPACC1  
Size: 12 Bytes  
Stack Space: 8 Bytes  
Calls: FLTADD  
Input: FPACC1 and FPACC2 contain the numbers to be subtracted.  
Output: FPACC1 contains the difference of the two numbers (FPACC1-FPACC2). FPACC2 remains unchanged.  
Error Conditions: Overflow, Underflow.  
Notes: Since FLTADD performs full signed addition, the floating-point subtract routine inverts the sign byte of FPACC2, calls FLTADD, and then changes the sign of FPACC2 back to what it was originally.

**FLOATING-POINT DIVIDE**

Subroutine Name: FLTDIV  
Operation: FPACC1 ÷ FPACC2 → FPACC1  
Size: 209 Bytes  
Stack Space: 11 Bytes  
Calls: PSHFPAC2, PULFPAC2  
Input: FPACC1 and FPACC2 contain the divisor and dividend respectively.  
Output: FPACC1 contains the quotient. FPACC2 remains unchanged.  
Error Conditions: Divide by zero, Overflow, Underflow

**FLOATING-POINT-TO-ASCII CONVERSION**

Subroutine Name: FLTASC  
Operation: FPACC1 → (X)  
Size: 370 Bytes  
Stack Space: 28 Bytes  
Calls: FLTMUL, FLTcmp, PSHFPAC2, PULFPAC2  
Input: FPACC1 contains the number to be converted to an ASCII string. The index register X points to a 14 byte string buffer.  
Output: The buffer pointed to by the X index register contains an ASCII string that represents the number in FPACC1. The string is terminated with a zero (0) byte and the X register points to the start of the string.  
Error Conditions: None
FLOATING POINT COMPARE

Subroutine Name: FLTCMP
Operation: FPACC1 - FPACC2
Size: 42 Bytes
Stack Space: None
Calls: None
Input: FPACC1 and FPACC2 contain the numbers to be compared.
Output: Condition codes are properly set so that all branch instructions may be used to alter program flow. FPACC1 and FPACC2 remain unchanged.
Error Conditions: None

UNSIGNED INTEGER TO FLOATING POINT

Subroutine Name: UINT2FLT
Operation: (16-bit unsigned integer) \* FPACC1
Size: 18 Bytes
Stack Space: 6 Bytes
Calls: FPNORM, CHK0
Input: The lower 16-bits of the FPACC1 mantissa contain an unsigned 16-bit integer.
Output: FPACC1 contains the floating-point representation of the 16-bit unsigned integer.
Error Conditions: None

SIGNED INTEGER TO FLOATING POINT

Subroutine Name: SINT2FLT
Operation: (16-bit signed integer) \* FPACC1
Size: 24 Bytes
Stack Space: 7 Bytes
Calls: UINT2FLT
Input: The lower 16-bits of the FPACC1 mantissa contain a signed 16-bit integer.
Output: FPACC1 contains the floating-point representation of the 16-bit signed integer.
Error Conditions: None

FLOATING POINT TO INTEGER

Subroutine Name: FLT2INT
Operation: FPACC1 \* (16-bit signed or unsigned integer)
Size: 74 Bytes
Stack Space: 2 Bytes
Calls: CHK0
Input: FPACC1 may contain a floating-point number in the range 65535 \( \leq \) FPACC1 \( \geq \) -32767.
Output: The lower 16-bits of the FPACC1 mantissa will contain a 16-bit signed or unsigned number.
Error Conditions: None
Notes: If the floating-point number in FPACC1 is positive, it will be converted to an unsigned integer. If the number is negative it will be converted to a signed two's complement integer. This type of conversion will allow 16-bit addresses to be represented as positive numbers in floating-point format. Any fractional part of the floating-point number is discarded.

TRANSFER FPACC1 TO FPACC2

Subroutine Name: TFR1TO2
Operation: FPACC1 \* FPACC2
Size: 13 Bytes
Stack Space: 0 Bytes
Calls: None
Input: FPACC1 contains a floating-point number.
Output: FPACC2 contains the same number as FPACC1.
Error Conditions: None
FLOATING-POINT FUNCTIONS

The following paragraphs describe the supplied floating-point functions, returned results, and possible error conditions. Note that even though the Taylor series which is used to calculate the trig functions requires that the input angle be expressed in radians; less precision is lost through angle reduction if the angle being reduced is expressed in degrees. Once the angle is reduced, the DEG2RAD subroutine is called to convert the angle to radians.

To reduce the number of factors in the Taylor expansion series all angles are reduced to fall between 0° and 45° by the ANGRED subroutine. This subroutine returns the reduced angle in FPACC1 along with the quad number that the original angle was in, and a flag that tells the calling routine whether it actually needs to calculate the sine or the cosine of the reduced angle to obtain the proper answer.

SQUARE ROOT

<table>
<thead>
<tr>
<th>Subroutine Name:</th>
<th>FLTSQR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation:</td>
<td>( \sqrt{FPACC1 \cdot FPACC1} )</td>
</tr>
<tr>
<td>Size:</td>
<td>104 Bytes</td>
</tr>
<tr>
<td>Stack Space:</td>
<td>21 Bytes</td>
</tr>
<tr>
<td>Calls:</td>
<td>TFR1TO2, FLTADD, FLTPAC2, PSHFPAC2, PULFPAC2</td>
</tr>
<tr>
<td>Input:</td>
<td>FPACC1 contains a valid floating-point number.</td>
</tr>
<tr>
<td>Output:</td>
<td>FPACC1 contains the square root of the original number. FPACC2 is unchanged.</td>
</tr>
<tr>
<td>Error Conditions:</td>
<td>NSQRTER is returned if the number in FPACC1 is negative and FPACC1 remains unchanged.</td>
</tr>
</tbody>
</table>

SINE

<table>
<thead>
<tr>
<th>Subroutine Name:</th>
<th>FLTSIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation:</td>
<td>SIN (FPACC1) ( \cdot FPACC1 )</td>
</tr>
<tr>
<td>Size:</td>
<td>380 Bytes (Includes SINCOS subroutine)</td>
</tr>
<tr>
<td>Stack Space:</td>
<td>50 Bytes</td>
</tr>
<tr>
<td>Calls:</td>
<td>ANGRED, SINCOS, DEG2RAD, PSHFPAC2, PULFPAC2</td>
</tr>
<tr>
<td>Input:</td>
<td>FPACC1 contains an angle in radians in the range (-2\pi \leq FPACC1 \leq +2\pi).</td>
</tr>
<tr>
<td>Output:</td>
<td>FPACC1 contains the sine of FPACC1, and FPACC2 remains unchanged.</td>
</tr>
<tr>
<td>Error Conditions:</td>
<td>None</td>
</tr>
<tr>
<td>Notes:</td>
<td>The Taylor Expansion Series is used to calculate the sine of the angle between 0° and 45° (( \pi/4 )). The subroutine ANGRED is called to reduce the input angle to within this range. Spot checks show a maximum error of ( +1.5 \times 10^{-7} ) throughout the input range.</td>
</tr>
</tbody>
</table>

COSINE

<table>
<thead>
<tr>
<th>Subroutine Name:</th>
<th>FLTCOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation:</td>
<td>COS(FPACC1) ( \cdot FPACC1 )</td>
</tr>
<tr>
<td>Size:</td>
<td>384 Bytes (Includes SINCOS subroutine)</td>
</tr>
<tr>
<td>Stack Space:</td>
<td>50 Bytes</td>
</tr>
<tr>
<td>Calls:</td>
<td>ANGRED, FLTSIN, DEG2RAD, PSHFPAC2</td>
</tr>
<tr>
<td>Input:</td>
<td>FPACC1 contains an angle in radians in the range (-2\pi \leq FPACC1 \leq +2\pi).</td>
</tr>
<tr>
<td>Output:</td>
<td>FPACC1 contains the cosine of FPACC1, and FPACC2 remains unchanged.</td>
</tr>
<tr>
<td>Error Conditions:</td>
<td>None</td>
</tr>
<tr>
<td>Notes:</td>
<td>The Taylor Expansion Series is used to calculate the cosine of the angle between 0° and 45° (( \pi/4 )). The subroutine ANGRED is called to reduce the input angle to within this range. Spot checks show a maximum error of ( +1.5 \times 10^{-7} ) throughout the input range.</td>
</tr>
</tbody>
</table>
TANGENT
Subroutine Name: FLTTAN
Operation: TAN(FPACC1) \* FPACC1
Size: 35 Bytes (Also requires FLTSIN and FLTCOS)
Stack Space: 56 Bytes
Calls: TFR1TOT2, EXG1AND2, FLTSIN, FLTCOS, FLTDMV, PSHFPAC2, PULFPAC2
Input: FPACC1 contains an angle in radians in the range \(-2\pi \leq \text{FPACC1} \leq +2\pi\).
Output: FPACC1 contains the tangent of the input angle, and FPACC2 remains unchanged.
Error Conditions: Returns largest legal number if tangent of \(\pm \pi / 2\) is attempted.
Notes: The tangent of the input angle is calculated by first obtaining the sine and cosine of the input angle and then using the following formula: \(\text{TAN} = \frac{\text{SIN}}{\text{COS}}\). At 89.9° the tangent function is only accurate to 5 decimal digits. For angles greater than 89.9° accuracy decreases rapidly.

DEGREES TO RADIANS CONVERSION
Subroutine Name: DEG2RAD
Operation: FPACC1 \* \pi - 180 \* FPACC1
Size: 15 Bytes
Stack Space: 16 Bytes
Calls: DEG2RAD
Input: Any valid floating-point number representing an angle in degrees.
Output: Input angles equivalent in radians.
Error Conditions: None

RADIANS TO DEGREES CONVERSION
Subroutine Name: RAD2DEG
Operation: FPACC1 \* 180 \div \pi \* FPACC1
Size: 8 Bytes (Also requires DEG2RAD subroutine)
Stack Space: 16 Bytes
Calls: DEG2RAD
Input: Any valid floating-point number representing an angle in radians.
Output: Input angles equivalent in degrees.
Error Conditions: Overflow, Underflow.

PI
Subroutine Name: GETPI
Operation: \(\pi\) \* FPACC1
Size: 6 Bytes
Stack Space: None
Input: None
Output: The value of \(\pi\) is returned in FPACC1.
Error Conditions: None
Notes: This routine should be used to obtain the value of \(\pi\) if it is required in calculations since it is accurate to the full 24 bits of the mantissa.
FORMAT CONVERSION Routines

As discussed in FLOATING-POINT ACCUMULATOR FORMAT and MEMORY FORMAT, the format for floating-point numbers as they appear in the floating-point accumulators is different than the way numbers are stored in memory. This was done primarily to save memory when a large number of floating-point variables are used in a program. Four routines are provided to convert to and from the different formats while at the same time moving a number into or out of the floating-point accumulators. By always using these routines to move numbers into and out of the floating-point accumulators, it would be extremely easy to adapt this floating-point package to work with any other floating-point format.

One example might be to interface this package with code produced by Motorola's 68HC11 'C' compiler. The Motorola 'C' compiler generates code for single-precision floating-point numbers whose internal format is that defined by the IEEE Standard for Binary Floating-Point Arithmetic. By rewriting the four routines described below the IEEE format could be easily converted to the format required by this floating-point package.

Get FPACC(x)

<table>
<thead>
<tr>
<th>Subroutine Name:</th>
<th>GETFPAC1 and GETFPAC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation:</td>
<td>(X) FPACC1; (X) FPACC2</td>
</tr>
<tr>
<td>Size:</td>
<td>22 Bytes each</td>
</tr>
<tr>
<td>Stack Space:</td>
<td>None</td>
</tr>
<tr>
<td>Input:</td>
<td>The X index register points to the 'memory formatted' number to be moved into the floating-point accumulator.</td>
</tr>
<tr>
<td>Output:</td>
<td>The number pointed to by X is in the specified floating-point accumulator.</td>
</tr>
<tr>
<td>Error Conditions:</td>
<td>None</td>
</tr>
</tbody>
</table>

Put FPACC(x)

<table>
<thead>
<tr>
<th>Subroutine Name:</th>
<th>PUTFPAC1 and PUTFPAC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation:</td>
<td>FPACC1; (X); FPACC2; (X)</td>
</tr>
<tr>
<td>Size:</td>
<td>22 Bytes each</td>
</tr>
<tr>
<td>Stack Space:</td>
<td>None</td>
</tr>
<tr>
<td>Input:</td>
<td>The X index register points to four consecutive memory locations where the number will be stored.</td>
</tr>
<tr>
<td>Output:</td>
<td>The floating point accumulator is moved into consecutive memory locations pointed to by the X index register.</td>
</tr>
<tr>
<td>Error Conditions:</td>
<td>None</td>
</tr>
</tbody>
</table>
The source code for this floating point package for the MC68HC11 * may be freely distributed under the rules of public domain. However * it is a copyrighted work and as such may not be sold as a product * or be included as part of a product for sale without the express * permission of the author. Any object code produced by the source * code may be included as part of a product for sale. * * If there are any questions or comments about the floating point * package please feel free to contact me.

Gordon Doughman
Motorola Semiconductor
3490 South Dixie Drive
Dayton, OH 45439
(513) 294-2231

---

ORG $0000
FPACCIEX RMB 1 FLOATING POINT ACCUMULATOR #1.
FPACC1MN RMB 3
MANTSGN1 RMB 1 MANTISSA SIGN FOR FPACC1 (0=-, FF=·).
FPACC2EX RMB 1 FLOATING POINT ACCUMULATOR #2.
FPACC2MN RMB 3
MANTSGN2 RMB 1 MANTISSA SIGN FOR FPACC2 (0=-, FF=·).

FLTPMTER EQU 1 /* floating point format error in ASCFLT */
OVFERR EQU 2 /* floating point overflow error */
UNFERR EQU 3 /* floating point underflow error */
DIVOERR EQU 4 /* division by 0 error */
TOLGSMER EQU 5 /* number too large or small to convert to int. */
NSORTERR EQU 6 /* tried to take the square root of negative # */
TAN900ERR EQU 7 /* TANgent of 90 degrees attempted */
This routine will accept most any ASCII floating point format and return a 32-bit floating point number. The following are some examples of legal ASCII floating point numbers.

- 20.095
- 0.125
- 7.2984E10
- 167.824E5
- 5.9357E-7
- 500

The floating point number returned is in "FPACC1".

The exponent is biased by 128 to facilitate floating point comparisons. A pointer to the ASCII string is passed to the routine in the D-register.

**LOCAL VARIABLES (ON STACK POINTED TO BY Y)**

- EXPSIGN EQU 0  EXPONENT SIGN (O=., FF;·).
- PWR10EXP EQU 1  POWER 10 EXPONENT.
- ORG $0000  (TEST FOR EVB)
- ASCFLT EQU *  SAVE POINTER TO ASCII STRING.
- PSHX PSHFPAC2  SAVE FPACC2.
- JSR PSXFPAC2  SAVE ZEROS ON STACK TO INITIALIZE LOCALS.
- LDX #0  ALLOCATE 2 BYTES FOR LOCALS.
- PSX STX FPACC1EX  CLEAR FPACC1.
- STX FPACC1EX+2  MAKE THE MANTISSA SIGN POSITIVE INITIALLY.
- CLR MANTISGN1  POINT TO LOCALS.
- LDX #6,Y  GET POINTER TO ASCII STRING.
- ASCFLT4 LDAA 0,X  GET 1ST CHARACTER IN STRING.
- JSR NUMERIC  IS IT A NUMBER.
- BCS ASCFLT4  YES, GO PROCESS IT.
- LEADING MINUS SIGN ENCOUNTERED?
- CMPA #1  NO, IS IT A MINUS SIGN?
0112 CO10 26 OB  BNE ASCFLT3  NO. GO CHECK FOR DECIMAL POINT.
0113 CO1F 73 00 04  BNE ASCFLT13  NO. GO CHECK FOR DECIMAL POINT.
0114 CO22 08  COM  MANTSIGN1  YES. SET MANTISSA SIGN. LEADING MINUS BEFORE?
0115 CO23 A6 00  INX  POINT TO NEXT CHARACTER.
0116 CO25 80 C1 55  LDA 0, X GET IT.
0117 CO28 25 19  JSR  NUMERIC  IS IT A NUMBER?
0118  *  *  LEADING DECIMAL POINT?
0119  *  *  FLOATING POINT FORMAT ERROR
0120  *  *  PRE DECIMAL POINT MANTISSA BUILD
0121  *  *  PRE DECIMAL POINT MANTISSA OVERFLOW
0122 CO2A 81 2E  CMPA #'.'  IS IT A DECIMAL POINT?
0123 CO2C 26 OB  BNE ASCFLT5  NO. FORMAT ERROR.
0124 CO2E 08  INX  YES. POINT TO NEXT CHARACTER.
0125 CO2F A6 00  LDA 0, X GET IT.
0126 CO31 80 C1 55  JSR  NUMERIC  MUST HAVE AT LEAST ONE DIGIT AFTER D.P.
0127 CO34 24 03  BCC ASCFLT15  GO REPORT ERROR.
0128 CO36 7E C0 C1  JMP ASCFLT11  GO BUILD FRACTION.
0129  *  *  LEADING DECIMAL POINT?
0130  *  *  FLOATING POINT FORMAT ERROR
0131  *  *  PRE DECIMAL POINT MANTISSA BUILD
0132 CO39 31  ASCFLT5  INS  DE-ALLOCATE LOCALS.
0133 CO3A 31  INS
0134 CO3B 80 C8 43  JSR PULFPAC2  RESTORE FPACC2.
0135 CO3E 38  PULX  GET POINTER TO TERMINATING CHARACTER IN STRING.
0136 CO3F 86 01  LDAA #$LTFMTER  FORMAT ERROR.
0137 CO41 00  SEC  SET ERROR FLAG.
0138 CO42 39  RTS  RETURN.
0139  *  *  PRE DECIMAL POINT MANTISSA BUILD
0140  *  *  PRE DECIMAL POINT MANTISSA OVERFLOW
0141  *  *  PRE DECIMAL POINT MANTISSA OVERFLOW
0142 CO43 A6 00  ASCFLT4  LDA 0, X
0143 CO45 80 C1 55  JSR  NUMERIC
0144 CO48 24 72  BCC ASCFLT10
0145 CO4A 80 C0 02  JSR ADDNXTD
0146 CO4D 08  INX
0147 CO4E 24 F3  BCC ASCFLT4
0148  *  *  PRE DECIMAL POINT MANTISSA OVERFLOW
0149  *  *  PRE DECIMAL POINT MANTISSA OVERFLOW
0150  *  *  PRE DECIMAL POINT MANTISSA OVERFLOW
0151 CO50 7C 00 00  ASCFLT6  INC  FPACC1EX  INC FOR EACH DIGIT ENCOUNTERED PRIOR TO D.P.
0152 CO53 A6 00  LDA 0, X GET NEXT CHARACTER.
0153 CO55 08  INX  POINT TO NEXT.
0154 CO56 80 C1 55  JSR  NUMERIC  IS IT A DIGIT?
0155 CO59 25 F5  BCS ASCFLT6  YES. KEEP BUILDING POWER 10 MANTISSA.
0156 CO5B 81 2E  CMPA '#'  NO. IS IT A DECIMAL POINT?
0157 CO5D 26 0A  BNE ASCFLT7  NO. GO CHECK FOR THE EXPONENT.
0158  *  *  ANY FRACTIONAL DIGITS ARE NOT SIGNIFICANT
0159  *  *  ANY FRACTIONAL DIGITS ARE NOT SIGNIFICANT
0160  *  *  ANY FRACTIONAL DIGITS ARE NOT SIGNIFICANT
0161 CO5F A6 00  ASCFLT8  LDA 0, X GET THE NEXT CHARACTER.
0162 CO61 80 C1 55  JSR  NUMERIC  IS IT A DIGIT?
0163 CO64 24 03  BCC ASCFLT7  NO. GO CHECK FOR AN EXPONENT.
0164 CO66 08  INX  POINT TO THE NEXT CHARACTER.
0165 CO67 20 F6  BRA ASCFLT18  FLUSH REMAINING DIGITS.
0166 CO69 B1 65  ASCFLT17  CMPA #'E'  NO. IS IT THE EXPONENT?
0167 CO6B 27 03  BEQ ASCFLT13  YES. GO PROCESS IT.
0168 CO6D 7E C1 17  JMP FINISH  NO. GO FINISH THE CONVERSION.
0169  *  *  PROCESS THE EXPONENT
0170  *  *  PROCESS THE EXPONENT
0171  *  *  PROCESS THE EXPONENT
0172 CO70 08  ASCFLT13  INX  POINT TO NEXT CHARACTER.
0173 CO71 A6 00  LDA 0, X GET THE NEXT CHARACTER.
0174 CO73 80 C1 55  JSR  NUMERIC  SEE IF IT'S A DIGIT.
0175 CO76 25 15  BCS ASCFLT9  YES. GET THE EXPONENT.
0176 CO78 B1 20  CMPA '#'  NO. IS IT A MINUS SIGN?
0177 CO7A 27 06  BEQ ASCFLT15  YES. GO FLAG A NEGATIVE EXPONENT.
0178 C07C 81 2B       CMPA   #*+ NO. IS IT A PLUS SIGN?
0179 C07E 27 05       BEQ ASCFLT16 YES, JUST IGNORE IT.
0180 C080 20 B7       BRA ASCFLT5 NO. FORMAT ERROR.
0181 C082 18 63 00     ASCFLT15 CMX EXPSIGN,Y FLAG A NEGATIVE EXPONENT. IS IT 1ST?
0182 C085 08          ASCFLT16 INX POINT TO NEXT CHARACTER.
0183 C086 A6 00       LDAA   0.X GET NEXT CHARACTER.
0184 C088 BD C1 55     JSR NUMERIC IS IT A NUMBER?
0185 C08B 24 AC       BCC ASCFLT5 NO. FORMAT ERROR.
0186 C08D 80 30       ASCFLT9 SUBA #34 MAKE IT BINARY.
0187 C08F 18 A7 01     STA PWR10EXP,Y BUILD THE POWER 10 EXPONENT.
0188 C092 0B 00       INX POINT TO NEXT CHARACTER.
0189 C093 A6 00       LDAA   0.X GET IT.
0190 C095 BD C1 55     JSR NUMERIC IS IT NUMERIC?
0191 C097 24 13       BCC ASCFLT14 NO. GO FINISH UP THE CONVERSION.
0192 C09A 18 E6 01     LDAB PWR10EXP,Y YES, GET PREVIOUS DIGIT.
0193 C09D 58          LSLB MULT. BY 2.
0194 C09E 58          LSLB NOW BY 4.
0195 C09F 18 EB 01     ADDB PWR10EXP,Y BY 5.
0196 C0A0 58          LSLB BY 10.
0197 C0A3 80 30       SUBA #33 MAKE SECOND DIGIT.
0198 C0A5 1B          ABA ADD IT TO FIRST DIGIT.
0199 C0A8 18 A7 01     STA PWR10EXP,Y IS THE EXPONENT OF RANGE?
0200 C0AB 01 26       CMPA  #35 IS THE EXPONENT OUT OF RANGE?
0201 C0A8 22 B6       BHI ASCFLT5 YES. REPORT ERROR.
0202 C0AD 18 A6 01     ASCFLT14 LDAB PWR10EXP,Y GET POWER 10 EXPONENT.
0203 C0B0 18 6D 00     TST EXPSIGN,Y WAS IT NEGATIVE?
0204 C0B3 2A 01       BPL ASCFLT12 NO. GO ADD IT TO BUILT 10 PWR EXPONENT.
0205 C0B5 40          NEG
0206 C0B6 98 00       ASCFLT12 ADDA FPACTEX FINAL TOTAL PWR 10 EXPONENT.
0207 C0B8 97 00       STA FPACTEX SAVE RESULT.
0208 C0BA 20 5B       BRA FINISH GO FINISH UP CONVERSION.
0209 *                * PRE-DECIMAL POINT NON-DIGIT FOUND, IS IT A DECIMAL POINT?
0210 *                *
0211 *                *
0212 C0BC 81 2E       ASCFLT10 CMPA  #'. IS IT A DECIMAL POINT?
0213 C0BE 26 A9       BNE ASCFLT17 NO. GO CHECK FOR THE EXPONENT.
0214 C0C0 08          INX YES. POINT TO NEXT CHARACTER.
0215 *                * POST DECIMAL POINT PROCESSING
0216 *                *
0217 *                *
0218 C0C1 A6 00       ASCFLT11 LDAA   0.X GET NEXT CHARACTER.
0219 C0C3 BD C1 55     JSR NUMERIC IS IT NUMERIC?
0220 C0C6 2A A1       BCC ASCFLT7 NO. GO CHECK FOR EXPONENT.
0221 C0C8 80 08       BSR ADDNXTD YES, ADD IN THE DIGIT.
0222 C0CA 08          INX POINT TO THE NEXT CHARACTER.
0223 C0CB 25 92       BCS ASCFLT8 IF OVER FLOW, FLUSH REMAINING DIGITS.
0224 C0CD 7A 00 00     DEC FPACTEX ADJUST THE 10 POWER EXPONENT.
0225 C0DE 20 EF       BRA ASCFLT11 PROCESS ALL FRACTIONAL DIGITS.
0226 *                *
0227 *                *
0228 *                *
0229 C0D0 96 01       ADDNXTD LDAA   FPACC1MN GET UPPER 8 BITS.
0230 C0D4 97 06       STA  FPACC2MN COPY INTO FPA2.
0231 C0D6 0C 02       LDD  FPACC1MN+1 GET LOWER 16 BITS OF MANTISSA.
0232 C0D8 00 07       STD  FPACC2MN+1 COPY INTO FPA2C.
0233 C0DA 05          LSIL MULT. BY 2.
0234 C0DB 79 00 01     ROL FPACC1MN OVERFLOW?
0235 C0DF 25 26       BCS ADDNXTD1 YES, DON'T ADD THE DIGIT IN.
0236 C0E0 05          LSIL MULT BY 4.
0237 C0E1 79 00 01     ROL FPACC1MN OVERFLOW?
0238 C0E4 25 26       BCS ADDNXTD1 YES, DON'T ADD THE DIGIT IN.
0239 C0E6 03 07       ADDF FPACC2MN+1 BY 5.
0240 C0E8 36          PSHA SAVE A.
0241 C0EA 96 01       LDAA   FPACC1MN GET UPPER 8 BITS.
0242 C0EB 89 00       ADCA   #0 ADDIN POSSIBLE CARRY FROM LOWER 16 BITS.
0243 C0ED 98 06       ADDA   FPACC2MN ADD IN UPPER 8 BITS.
BY 10.

NOW FINISH UP CONVERSION BY MULTIPLYING THE RESULTANT MANTISSA

BY 10 FOR EACH POSITIVE POWER OF 10 EXPONENT RECEIVED OR BY .1

(DIVIDE BY 10) FOR EACH NEGATIVE POWER OF 10 EXPONENT RECEIVED.

FINISH3 EQU * 

SAVE POINTER TO TERMINATING CHARACTER IN STRING.

POINT TO FPACC2.

EXIT.

FINISH1: 

GET CURRENT POWER 10 EXPONENT.

GETFPAC2: 

GET CONSTANT .1 (DIVIDE BY 10).

FINISH1: 

SET UP TERMINATING CHARACTER IN STRING.

FINISH1: 

GET CONSTANT INTO FPACC2.

STAX FPACC1MN SAVE IT.

PULX RESTORE A.

BCS ADDNEXTD1 OVERFLOW? IF SO DON'T ADD IT IN.

LSRD BY 10.

FPACC1MN+1 SAVE THE LOWER 16 BITS.

BCS ADDNEXTD1 OVERFLOW? IF SO DON'T ADD IT IN.

LDAB 0,x GET CURRENT DIGIT.

SUBB #$30 MAKE IT BINARY.

16-BIT.

ADD FPACC1MN+1 ADD IT IN TO TOTAL.

STD FPACC1MN+1 SAVE THE RESULT.

LDAA FPACC1MN GET UPPER 8 BITS.

ADDCA #0 ADD IN POSSIBLE CARRY, OVERFLOW?

BCS ADDNEXTD1 YES. COPY OLD MANTISSA FROM FPACC2.

STAA FPACC1MN NO. EVERYTHING OK.

RETURN.

ADDNEXTD1 LOAD FPACC2MN+1 RESTORE THE ORIGINAL MANTISSA BECAUSE

OF OVERFLOW.

STDA FPACC1MN+1 OF OVERFLOW.

RETURN.

* * *

NOW FINISH UP CONVERSION BY MULTIPLYING THE RESULTANT MANTISSA

BY 10 FOR EACH POSITIVE POWER OF 10 EXPONENT RECEIVED OR BY .1

(DIVIDE BY 10) FOR EACH NEGATIVE POWER OF 10 EXPONENT RECEIVED.

FINISH3 EQU *

SAVE POINTER TO TERMINATING CHARACTER IN STRING.

POINT TO FPACC2.

EXIT.

FINISH1: 

GET CURRENT POWER 10 EXPONENT.

GETCONST1 INTO FPACC2.

GETFPAC2: 

GET CONSTANT .1 (DIVIDE BY 10).

FINISH1: 

SET UP TERMINATING CHARACTER IN STRING.

FINISH1: 

GET CONSTANT INTO FPACC2.

STAX FPACC1MN SAVE IT.

PULX RESTORE A.

BCS ADDNEXTD1 OVERFLOW? IF SO DON'T ADD IT IN.

LSRD BY 10.

FPACC1MN+1 SAVE THE LOWER 16 BITS.

BCS ADDNEXTD1 OVERFLOW? IF SO DON'T ADD IT IN.

LDAB 0,x GET CURRENT DIGIT.

SUBB #$30 MAKE IT BINARY.

16-BIT.

ADD FPACC1MN+1 ADD IT IN TO TOTAL.

STD FPACC1MN+1 SAVE THE RESULT.

LDAA FPACC1MN GET UPPER 8 BITS.

ADDCA #0 ADD IN POSSIBLE CARRY, OVERFLOW?

BCS ADDNEXTD1 YES. COPY OLD MANTISSA FROM FPACC2.

STAA FPACC1MN NO. EVERYTHING OK.

RETURN.

ADDNEXTD1 LOAD FPACC2MN+1 RESTORE THE ORIGINAL MANTISSA BECAUSE

OF OVERFLOW.

STDA FPACC1MN+1 OF OVERFLOW.

RETURN.

* * *

NOW FINISH UP CONVERSION BY MULTIPLYING THE RESULTANT MANTISSA

BY 10 FOR EACH POSITIVE POWER OF 10 EXPONENT RECEIVED OR BY .1

(DIVIDE BY 10) FOR EACH NEGATIVE POWER OF 10 EXPONENT RECEIVED.

FINISH3 EQU *

SAVE POINTER TO TERMINATING CHARACTER IN STRING.

POINT TO FPACC2.

EXIT.

FINISH1: 

GET CURRENT POWER 10 EXPONENT.

GETCONST1 INTO FPACC2.

GETFPAC2: 

GET CONSTANT .1 (DIVIDE BY 10).

FINISH1: 

SET UP TERMINATING CHARACTER IN STRING.

FINISH1: 

GET CONSTANT INTO FPACC2.

STAX FPACC1MN SAVE IT.

PULX RESTORE A.

BCS ADDNEXTD1 OVERFLOW? IF SO DON'T ADD IT IN.

LSRD BY 10.

FPACC1MN+1 SAVE THE LOWER 16 BITS.

BCS ADDNEXTD1 OVERFLOW? IF SO DON'T ADD IT IN.

LDAB 0,x GET CURRENT DIGIT.

SUBB #$30 MAKE IT BINARY.

16-BIT.

ADD FPACC1MN+1 ADD IT IN TO TOTAL.

STD FPACC1MN+1 SAVE THE RESULT.

LDAA FPACC1MN GET UPPER 8 BITS.

ADDCA #0 ADD IN POSSIBLE CARRY, OVERFLOW?

BCS ADDNEXTD1 YES. COPY OLD MANTISSA FROM FPACC2.

STAA FPACC1MN NO. EVERYTHING OK.

RETURN.

ADDNEXTD1 LOAD FPACC2MN+1 RESTORE THE ORIGINAL MANTISSA BECAUSE

OF OVERFLOW.

STDA FPACC1MN+1 OF OVERFLOW.

RETURN.

* * *

NOW FINISH UP CONVERSION BY MULTIPLYING THE RESULTANT MANTISSA

BY 10 FOR EACH POSITIVE POWER OF 10 EXPONENT RECEIVED OR BY .1

(DIVIDE BY 10) FOR EACH NEGATIVE POWER OF 10 EXPONENT RECEIVED.

FINISH3 EQU *

SAVE POINTER TO TERMINATING CHARACTER IN STRING.

POINT TO FPACC2.
NUMERIC CLC NON-NUMERIC CHARACTER, CLEAR THE CARRY.
RTS RETURN.

FPNORM EQU *

LDX #FPACC1EX POINT TO FPACC1.
BSR CHK0 CHECK TO SEE IF IT'S 0.
BEQ FPNORM3 YES, JUST RETURN.
TST FPACC1MN IS THE NUMBER ALREADY NORMALIZED?
BMI FPNORM3 YES, JUST RETURN...

FPNORM1 LDD FPACC1MN+1 GET THE LOWER 16 BITS OF THE MANTISSA.
DEC FPACC1EX DECREMENT THE EXPONENT FOR EACH SHIFT.
BEQ FPNORM4 EXPONENT WENT TO 0, UNDERFLOW.
LSLD SHIFT THE LOWER 16 BITS.
ROL FPACC1MN ROTATE THE UPPER 8 BITS, NUMBER NORMALIZED?
BPL FPNORM2 NO, KEEP SHIFTING TO THE LEFT.
STD FPACC1MN+1 PUT THE LOWER 16 BITS BACK INTO FPACC1.

FPNORM3 CLC SHOW NO ERRORS.
RTS YES, RETURN.

FPNORM4 SEC FLAG ERROR.
RTS RETURN.

CHECK0 EQU * CHECKS FOR ZERO IN FPACC POINTED TO BY X.
PSHB SAVE D.
PSHA
LDD 0,X GET FPACC EXPONENT & HIGH 8 BITS.
BNE CHK01 NOT ZERO, RETURN.
LDD 2,X CHECK LOWER 16 BITS.
PULA RESTORE D.
PULB

RTS RETURN WITH CC SET.

CONST1 FCB $7D,$4C,$CC,$CD 0.1 DECIMAL
CONST10 FCB $84,$20,$00,$00 10.0 DECIMAL

*
**FPMULT: FLOATING POINT MULTIPLY**

- **THIS FLOATING POINT MULTIPLY ROUTINE MULTIPLIES "FPACC1" BY**
- "FPACC2" AND PLACES THE RESULT IN TO FPACC1. FPACC2 REMAINS**
- UNCHANGED.
- **WORSE CASE = 2319 CYCLES = 1159 US @ 2MHz**

---

**FLTMUL**

**-**

**FLTMUL**

**EQU**

**JSR** PSHFPACZ  SAVE FPACC2.

**LDA** #FPACC1EX POINT TO FPACC1

**JSR** CHKXO CHECK TO SEE IF FPACC1 IS ZERO.

**BEO** FPMULT3 IT IS. ANSWER IS 0.

**LDA** #FPACC2EX POINT TO FPACC2.

**JSR** CHKXO IS IT 0?

**BNE** FPMULT4 NO. CONTINUE.

**CLRA** CLEAR D.

**CLRB**

**STD** FPACC1EX MAKE FPACC1 0.

**STD** FPACC1MN+1

**BRA** FPMULT3 RETURN.

**FPMULT4** LDAEX MANTSGN1 GET FPACC1 EXPONENT.

**EORA** MANTSGN2 SET THE SIGN OF THE RESULT.

**STA** MANTSGN1 SAVE THE SIGN OF THE RESULT.

**LDA** FPACC1EX GET FPACC1 EXPONENT.

**ADD** FPACC2EX ADD IT TO FPACC2 EXPONENT.

**BPL** FPMULT1 IF RESULT IS MINUS AND

**BCC** FPMULT2 THE CARRY IS SET THEN:

**FPMULT5** LDAEX #OFERR OVERFLOW ERROR.

**SEC** SET ERROR FLAG.

**SEC**

**FPMULT6** LDAEX #OFERR RETURN.

**BRA** FPMULT6 RETURN.

**FPMULT1** BCS FPMULT2 IF RESULT IS PLUS & THE CARRY IS SET THEN ALL OK.

**LDAEX** #OFERR ELSE UNDERFLOW ERROR OCCURED.

**SEC** SET FLAG ERROR.

**SEC**

**FPMULT2** LDAEX ADD #80 ADD 128 BIAS BACK IN THAT WE LOST.

**STA** FPACC1EX SAVE THE NEW EXPONENT.

**JSR** UMULT GO MULTIPLY THE "INTEGER" MANTISSAS.

**FPMULT3** TST FPACC1EX WAS THERE AN OVERFLOW ERROR FROM Rounding?

**BEO** FPMULT5 YES. RETURN ERROR.

**CLC** CLEAR O.

**FPMULT6** JSR PULFPACZ RESTORE FPACC2.

**RTS**

**FPMULT**

**-**

**UMULT**

**EQU**

**LDA** #0 CREATE PARTIAL PRODUCT REGISTER AND COUNTER.

**PSHX**

**PSHX**

**TSX** POINT TO THE VARIABLES.

**LDAEX** #24 SET COUNT TO THE NUMBER OF BITS.

**STA** 0,X

**LDAEX** FPACC2MN+2 GET THE L.S. BYTE OF THE MULTIPLIER.

**LSRA** PUT L.S. BIT IN CARRY.

**BCC** UMULT2 IF CARRY CLEAR, DON'T ADD MULTIPLICAND TO P.P.

**LDA** FPACC1MN+1 GET MULTIPICAND L.S. 16 BITS.

**ADD** 2,X ADD TO PARTIAL PRODUCT.

**STD** 2,X SAVE IN P.P.

**LDA** FPACC1MN GET UPPER 8 BITS OF MULTIPLICAND.

**ADCA** 1,X ADD IT W/ CARRY TO P.P.
0410 C1F2 A7 01  
0411 ClF4 66 01  
0412 ClF6 66 02  
0413 ClF8 66 03  
0414 C1F4 76 00 06  
0415 ClF6 76 00 07  
0416 ClF8 76 00 08  
0417 C203 6A 00  
0418 C205 26 0C  
0419 C207 60 01  
0420 C209 2B 0C  
0421 C20B 78 00 06  
0422 C20D 69 03  
0423 C210 69 02  
0424 C212 69 01  
0425 C214 78 00 00  
0426 C217 7D 00 06  
0427 C21A 2A 1B  
0428 C21C EC 02  
0429 C21E C3 00 01  
0430 C221 ED 02  
0431 C223 AD 01  
0432 C225 89 00  
0433 C227 A7 01  
0434 C229 24 09  
0435 C22B 66 01  
0436 C22D 66 02  
0437 C22F 66 03  
0438 C231 7C 00 00  
0439 C234 31  
0440 C236 3B  
0441 C238 DF 01  
0442 C23B 32  
0443 C23D 97 03  
0444 C23F 39  
0445       
0446       
0447       

0410 ClF2 A7 01  
0411 ClF4 66 01  
0412 ClF6 66 02  
0413 ClF8 66 03  
0414 ClF4 76 00 06  
0415 ClF6 76 00 07  
0416 ClF8 76 00 08  
0417 C203 6A 00  
0418 C205 26 0C  
0419 C207 60 01  
0420 C209 2B 0C  
0421 C20B 78 00 06  
0422 C20D 69 03  
0423 C210 69 02  
0424 C212 69 01  
0425 C214 78 00 00  
0426 C217 7D 00 06  
0427 C21A 2A 1B  
0428 C21C EC 02  
0429 C21E C3 00 01  
0430 C221 ED 02  
0431 C223 AD 01  
0432 C225 89 00  
0433 C227 A7 01  
0434 C229 24 09  
0435 C22B 66 01  
0436 C22D 66 02  
0437 C22F 66 03  
0438 C231 7C 00 00  
0439 C234 31  
0440 C236 3B  
0441 C238 DF 01  
0442 C23B 32  
0443 C23D 97 03  
0444 C23F 39  
0445       
0446       
0447       

STAA 1,X  
ROR 1,X  
ROR 2,X  
ROR 3,X  
ROR FPACC2MN  
DEC 0,X  
BNE UMULT1  
TST 1,X  
BMI UMULT3  
LSL FPACC2MN  
ROL 3,X  
ROL 2,X  
ROL 1,X  
DEC FPACC1EX  
TST FPACC2MN  
LDD 2,X  
ADD #1  
STD 2,X  
LDAAD 1,X  
ADCA #0  
STAA 1,X  
BCC UMULT4  
ROR 1,X  
ROR 2,X  
ROR 3,X  
INC FPACC1EX  
PULX  
STX FPACC1MN  
PULX  
STAA FPACC1MN+2  
RTS  
SAVE TO PARTIAL PRODUCT.  
ROTATE PARTIAL PRODUCT TO THE RIGHT.  
SHIFT THE MULTIPLIER TO THE RIGHT 1 BIT.  
DONE YET?  
NO. KEEP GOING.  
DOES PARTIAL PRODUCT NEED TO BE NORMALIZED?  
NO. GET ANSWER & RETURN.  
GET BIT THAT WAS SHIFTED OUT OF P.P REGISTER.  
PUT IT BACK INTO THE PARTIAL PRODUCT.  
FIX EXPONENT.  
DO WE NEED TO ROUND THE PARTIAL PRODUCT?  
NO. JUST RETURN.  
YES. GET THE LEAST SIGNIFICANT 16 BITS.  
ADD 1.  
SAVE RESULT.  
PROPAGATE THROUGH.  
IF CARRY CLEAR ALL IS OK.  
IF NOT OVERFLOW. ROTATE CARRY INTO P.P.  
UP THE EXponent.  
TAKE COUNTER OFF STACK.  
GET M.S. 16 BITS OF PARTIAL PRODUCT.  
PUT IT IN FPACC1.  
GET L.S. 8 BITS OF PARTIAL PRODUCT.  
PUT IT IN FPACC1.  
RETURN.
This subroutine performs floating point addition of the two numbers in FPACC1 and FPACC2. The result of the addition is placed in FPACC1 while FPACC2 remains unchanged. This subroutine performs full signed addition so either number may be of the same or opposite sign.

WORSE CASE = 1030 CYCLES = .5 15 US @ 2 MHz

FLTADD EQU *

FLTADD: JSR PSHFPAC2 SAVE FPACC2.
LDX #FPACC2EX POINT TO FPACC2.
JSR CHK0 IS IT ZERO?
BNE FLTADD1 NO. GO CHECK FOR 0 IN FPACC1.

FLTADD6 CLC NO ERRORS.
JSR PULFPAC2 RESTORE FPACC2.

RTS ANSWER IN FPACC1. RETURN.

FLTADD1: LDX #FPACC1EX POINT TO FPACC1.
JSR CHK0 IS IT ZERO?
BNE FLTADD2 NO. GO ADD THE NUMBER.

LDOD FPACC2EX ANSWER IS IN FPACC2. MOVE IT INTO FPACC1.

STD FPACC1EX.
LDOD FPACC2MN+1 MOVE LOWER 16 BITS OF MANTISSA.
STD FPACC1MN+1.

LDAA MANTSGN2 MOVE FPACC2 MANTISSA SIGN INTO FPACC1.
STAA MANTSGN1.
BRA FLTADD6 RETURN.

LDAA FPACC1EX GET FPACC1 EXPONENT.
CMPA FPACC2MN ARE THE EXPONENTS THE SAME?
BEQ FLTADD7 YES. GO ADD THE MANTISSA'S.

SUBA FPACC2EX NO. FPACC1EX< FPACC2EX. IS FPACC1 > FPACC2?

BPL FLTADD3 YES. GO CHECK RANGE.

NEG FPACC1 MN+1 NO. FPACC1 < FPACC2. MAKE DIFFERENCE POSITIVE.

CMPA #23 ARE THE NUMBERS WITHIN RANGE?

BHI FLTADD4 NO. FPACC2 IS LARGER. GO MOVE IT INTO FPACC1.

TAB PUT DIFFERENCE IN B.

ADDB FPACC1EX CORRECT FPACC1 EXPONENT.

ADDB FPACC1EX SAVE THE RESULT.

LDX #FPACC1MN POINT TO FPACC1 MANTISSA.
BRA FLTADD5 GO DENORMALIZE FPACC1 FOR THE ADD.

CMPA #23 FPACC1 > FPACC2. ARE THE NUMBERS WITHIN RANGE?

BHI FLTADD6 NO. ANSWER ALREADY IN FPACC1. JUST RETURN.

LDX #FPACC2MN POINT TO THE MANTISSA TO DENORMALIZE.

LDR 0,X SHIFT THE FIRST BYTE OF THE MANTISSA.
ROR 1,X THE SECOND.
ROR 2,X AND THE THIRD.
DECA DONE YET?

BNE FLTADD5 NO. KEEP SHIFTING.

LDAA MANTSGN1 GET FPACC1 MANTISSA SIGN.
CMPA MANTSGN2 ARE THE SIGNS THE SAME?

BEQ FLTADD11 YES. JUST GO ADD THE TWO MANTISSAS.

TST MANTSGN1 NO. IS FPACC1 THE NEGATIVE NUMBER?

BPL FLTADD8 NO. GO DO FPACC1- FPACC2.

LDX FPACC2MN YES. EXCHANGE FPACC1 & FPACC2 BEFORE THE SUB.

PSHX SAVE IT.

LDX FPACC1MN GET PART OF FPACC1.
STX FPACC2MN PUT IT IN FPACC2.
PULX GET SAVED PORTION OF FPACC2.
STX FPACC1MN PUT IT IN FPACC1.

338
LDX FPACC2MN+2
GET LOWER 8 BITS & SIGN OF FPACC2.

PSHX SAVE IT.

LDX FPACC1MN+2
GET LOWER 8 BITS & SIGN OF FPACC1.

STX FPACC2MN+2
PUT IT IN FPACC2.

PULX GET SAVED PART OF FPACC2.

STX FPACC1MN+2
PUT IT IN FPACC1.

FLTADO8 LDD FPACC1MN+1
GET LOWER 16 BITS OF FPACC1.

SUBD FPACC2MN+1
SUBTRACT LOWER 16 BITS OF FPACC2.

STD FPACC1MN+1
SAVE RESULT.

LOADA FPACC1MN
GET HIGH 8 BITS OF FPACC1 MANTISSA.

SBCA FPACC2MN
SUBTRACT HIGH 8 BITS OF FPACC2.

STAA FPACC1MN
SAVE THE RESULT. IS THE RESULT NEGATIVE?

COMA
SAVE THE RESULT.

COMA
FORM THE ONE'S COMPLEMENT.

COMA
FORM THE TWO'S COMPLEMENT.

ADD#1
SUBTRACT UPPER 8 BITS BACK.

PULA
ADD IN POSSIBLE CARRY.

ADCA #0
SAVE RESULT.

LOADA #FFFF
SHOW THAT FPACC1 IS NEGATIVE.

STAA MANTISSA1
GO NORMALIZE THE RESULT.

BCC FLTADD12
EVERYTHING'S OK SO RETURN.

BCC FLTADD10
RETURN.

LEAD
ADD IT TO THE LOWER 16 BITS OF FPACC2.

LOAD#ADDM1
ADD RESULT IN FPACC1.

LOADA FPACC1MN
GET UPPER 8 BITS OF FPACC1.

ADCA FPACC2MN
ADD IT (WITH CARRY) TO UPPER 8 BITS OF FPACC2.

STAA FPACC1MN
SAVE THE RESULT.

BCC FLTADD12
NO OVERFLOW SO JUST RETURN.

ROR FPACC1MN
PUT THE CARRY INTO THE MANTISSA.

ROR FPACC1MN+1
PROPAGATE THROUGH MANTISSA.

INC FPACC1EX
UP THE MANTISSA BY 1.

BNE FLTADD12
EVERYTHING'S OK JUST RETURN.

LDDA #OVFERR
RESULT WAS TOO LARGE, OVERFLOW.

SEC FLAG ERROR.

JMP FLTADD10
RETURN.

*
**Floating Point Subtract Subroutine**

This subroutine performs floating point subtraction (FPACC1-FPACC2) by inverting the sign of FPACC2 and then calling FLTADD since FLTADD performs complete signed addition. Upon returning from FLTADD the sign of FPACC2 is again inverted to leave it unchanged from its original value.

**Worse Case = 1062 Cycles = 531 μs @ 2MHz**

---

**TTL FLT SUB**

---

```
0564   TTL   FLT SUB
0565   ******************************************************
0566   *
0567   * FLOATING POINT SUBTRACT SUBROUTINE
0568   *
0569   * This subroutine performs floating point subtraction (FPACC1-FPACC2)
0570   * by inverting the sign of FPACC2 and then calling FLTADD since
0571   * FLTADD performs complete signed addition. Upon returning from
0572   * FLTADD the sign of FPACC2 is again inverted to leave it unchanged
0573   * from its original value.
0574   *
0575   * WORSE CASE = 1062 CYCLES = 531 μs @ 2MHz
0576   *
0577   *
0578   *
0579   *
0580   C2FE  
0581   C2FE  B0 03  
0582   C300  B0 C2 3C  
0583   C303  96 09  
0584   C305  B8 FF  
0585   C307  97 09  
0586   C309  39  
0587   *
0588   *
0589   *
```

---

```
0580 C2FE
0581 C2FE B0 03
0582 C300 B0 C2 3C
0583 C303 96 09
0584 C305 B8 FF
0585 C307 97 09
0586 C309 39
```

---

```
0580 C2FE  
0581 C2FE  B0 03  
0582 C300  B0 C2 3C  
0583 C303  96 09  
0584 C305  B8 FF  
0585 C307  97 09  
0586 C309  39  
```

---

```
0580 C2FE
0581 C2FE B0 03
0582 C300 B0 C2 3C
0583 C303 96 09
0584 C305 B8 FF
0585 C307 97 09
0586 C309 39
```

---

```
0580 C2FE
0581 C2FE B0 03
0582 C300 B0 C2 3C
0583 C303 96 09
0584 C305 B8 FF
0585 C307 97 09
0586 C309 39
```

---

```
0580 C2FE
0581 C2FE B0 03
0582 C300 B0 C2 3C
0583 C303 96 09
0584 C305 B8 FF
0585 C307 97 09
0586 C309 39
```
This subroutine performs signed floating point divide. The operation performed is FPACC1/FPACC2. The divisor (FPACC2) is left unaltered and the answer is placed in FPACC1. There are several error conditions that can be returned by this routine. They are:

a) division by zero. b) overflow. c) underflow. As with all other routines, an error is indicated by the carry being set and the error code being in the A-reg.

WORSE CASE = 2911 CYCLES = 1455 us @ 2MHz

IIORSE CASE = 2911 CYCLES = 1455 us @ 2MHz

FLTDIV EQU *
LDX #FPACC2EX POINT TO FPACC2.
JSR CHCK0 IS THE DIVISOR O?
BNE FLTDIV1 NO. GO SEE IF THE DIVIDEND IS ZERO.
LDAA #DIVOERR YES. RETURN A DIVIDE BY ZERO ERROR.
SEC FLAG ERROR.
RTS RETURN.
FLTDIV1 LDX #FPACC1EX POINT TO FPACC1.
JSR CHCK0 IS THE DIVIDEND O?
BNE FLTDIV2 NO. GO PERFORM THE DIVIDE.
CLC YES. ANSWER IS ZERO. NO ERRORS.
RTS RETURN.
FLTDIV2 JSR PSHFPAC2 SAVE FPACC2.
LDAA MANTSGN2 GET FPACC2 MANTISSA SIGN.
EORA MANTSGNl SET THE SIGN OF THE RESULT.
STAA MANTSGNl SAVE THE RESULT.
LOX #D SET UP 10RK SPACE ON THE STACK.
PSHX PSHX PSHX
LDAA #24 PUT LOOP COUNT ON STACK.
PSHX
LOAA #24 PUT LOOP COUNT ON STACK.
PSHA
TSX SET UP POINTER TO WORK SPACE.
LDD FPACC1MN COMPARE FPACC1 & FPACC2 MANTISSAS.
CPO FPACC2MN ARE THE UPPER 16 BITS THE SAME?
BNE FLTDIV3 NO.
LDAA FPACC1MN+2 YES. COMPARE THE LOWER 8 BITS.
CMPA FPACC2MN+2 FLTDIV4 IS FPACC2 MANTISSA > FPACC1 MANTISSA? NO.
INC FPACC2EX ADD 1 TO THE EXPONENT TO KEEP NUMBER THE SAME.
BNE FLTDIV14 NO. GO SHIFT THE MANTISSA RIGHT 1 BIT.
LDAA #MOVERR YES. GET ERROR CODE.
SEC FLAG ERROR.
FLTDIV6 PULX REMOVE WORKSPACE FROM STACK.
PULX PULX
INS
LDD FPACC1MN+1 DO AN INITIAL SUBTRACT IF DIVIDEND MANTISSA IS GREATER THAN DIVISOR MANTISSA.
BNE FLTDIV14 NO. GO SHIFT THE MANTISSA RIGHT 1 BIT.
LDAA FPACC2MN+1 GREATER THAN DIVISOR MANTISSA.
STD FPACC1MN+1
LOAA FPACC1MN
SBCA FPACC2MN
STAAC FPACC1MN
DEC O,X SUBTRACT 1 FROM THE LOOP COUNT.
PULFPAC2 RESTORE FPACC2.
RTS RETURN.
PULX
LDX #FPACC1MN+1 DO AN INITIAL SUBTRACT IF DIVIDEND MANTISSA IS GREATER THAN DIVISOR MANTISSA.
STD FPACC1MN+1
LOAA FPACC1MN
SBCA FPACC2MN
STAAC FPACC1MN
DEC O,X SUBTRACT 1 FROM THE LOOP COUNT.

055 C3E 74 00 06
056 C3E 76 00 07
057 C3E 76 00 08
058 C3E 96 00
059 C3E 08 05
060 C3E 58 50
061 C3E 18
062 C3E 28 06
063 C3E 25 06
064 C3E 86 03
065 C3E 20 03
066 C3E 25 CE
067 C3E 88 81
068 C3E 97 00
069 C3E 0C 01
070 C3E E0 04
071 C3E 96 03
072 C3E A7 06
073 C3E 0C 02
074 C3E 93 07
075 C3E 0D 02
076 C3E 96 01
077 C3E 92 06
078 C3E 97 01
079 C3F 2A 08
080 C3E 94 04
081 C3E 00 01
082 C3E A6 06
083 C3E 97 03
084 C3E 69 03
085 C3E 69 02
086 C3E 69 01
087 C3F 78 00 03
088 C3F 79 00 02
089 C3F 79 00 01
090 C3E 6A 00
091 C3E 26 CF
092 C3E 63 01
093 C3E 63 02
094 C3E 63 03
095 C3F 96 02
096 C3F 93 07
097 C3F 96 01
098 C3F 92 06
099 C3F 94 02
100 C3E 24 03
101 C3E 0C
102 C3F 20 03
103 C3E 0C 02
104 C3E 0C 01
105 C3E 0C 00 01
106 C3E 89 0C
107 C3E 97 01
108 C3E 2C 09
109 C3E 70 0C 01
110 C3E 76 00 02
111 C3E 76 00 03
112 C3E 0C
113 C3E 7E C3 48
114 *
115 *
116 *

065 CS E74 00 06
066 CS E76 00 07
067 CS E76 00 08
068 CS E96 00
069 CS E08 05
070 CS E58 50
071 CS E18
072 CS E28 06
073 CS E25 06
074 CS E86 03
075 CS E20 03
076 CS E25 CE
077 CS E88 81
078 CS E97 00
079 CS E0C 01
080 CS E40 04
081 CS E96 03
082 CS EA7 06
083 CS EC0 02
084 CS E93 07
085 CS E0D 02
086 CS E96 01
087 CS E92 06
088 CS E94 02
089 CS E0C
090 CS E6A 00
091 CS E26 CF
092 CS E63 01
093 CS E63 02
094 CS E63 03
095 CS F96 02
096 CS F93 07
097 CS F96 01
098 CS F92 06
099 CS F94 02
100 CS F24 03
101 CS F0C
102 CS F20 03
103 CS F0C 02
104 CS F0C 01
105 CS F0C 00 01
106 CS E89 0C
107 CS E97 01
108 CS E2C 09
109 CS E70 0C 01
110 CS E76 00 02
111 CS E76 00 03
112 CS E0C
113 CS E7E C3 48
114 *
115 *
116 *

FLTDIV14 LSR FPAC2MN
ROR FPAC2MN+1
ROR FPAC2MN+2
LDA FPACC1EX GET FPACC1 EXPONENT.
LDA FPACC2EX GET FPACC2 EXPONENT.
NEGB ADD THE TWO'S COMPLEMENT TO SET FLAGS PROPERLY.
ABA
BMI FLTDIV5 IF RESULT MINUS CHECK CARRY FOR POSS. OVERFLOW.
BCS FLTDIV7 IF PLUS & CARRY SET ALL IS OK.
LDAA #0NFERR IF NOT, UNDERFLOW ERROR.
BRA FLTDIV6 RETURN WITH ERROR.
FLTDIV5 BCS FLTDIV8 IF MINUS & CARRY SET OVERFLOW ERROR.
FLTDIV7 ADDA #581 ADD BACK BIAS+1 (THE '1' COMPENSATES FOR ALG.)
STAA FPACC1EX SAVE RESULT.
FLTDIV9 LDD FPACC1MN SAVE DIVIDEND IN CASE SUBTRACTION DOESN'T GO.
STD 4, X
LDA FPACC1MN+2
STAA 6, X
LDD FPACC1MN+1 GET LOWER 16 BITS FOR SUBTRACTION.
SUBB FPACC2MN+1
STD FPACC1MN+1 SAVE RESULT.
LDAA FPACC1MN GET HIGH 8 BITS.
SBCA FPACC2MN
STAA FPACC1MN
FLTDIV10 ROL 3, X ROTATE CARRY INTO QUOTIENT.
ROR 2, X
ROR 1, X
LSL FPACC1MN+2 SHIFT DIVIDEND TO LEFT FOR NEXT SUBTRACT.
ROL FPACC1MN+1
ROL FPACC1MN
DEC 0, X DONE YET?
SNE FLTDIV9 NO, KEEP GOING.
COM 1, X RESULT MUST BE COMPLEMENTED.
COM 2, X
COM 3, X
LDD FPACC1MN+1 DO 1 MORE SUBTRACT FOR Rounding.
SUBB FPACC2MN+1 (? DON'T NEED TO SAVE THE RESULT.)
LDAA FPACC1MN
SBCA FPACC2MN (NO NEED TO SAVE THE RESULT.)
LDD 2, X GET LOW 16 BITS.
FLTDIV11 BCC FLTDIV7 IF IT DIDN'T GO RESULT OK AS IS.
FLTDIV12 CLC CLEAR THE CARRY.
FLTDIV13 BRA FLTDIV13 GO SAVE THE NUMBER.
FLTDIV14 ADDD #1 ROUND UP BY 1.
FLTDIV15 STD FPACC1MN+1 PUT IT IN FPACC1.
LDAA 1, X GET HIGH 8 BITS.
AGCA #0
FLTDIV16 CLA NO ERRORS.
JMP FLTDIV6 RETURN.
FLOATING POINT TO ASCII CONVERSION SUBROUTINE

This subroutine performs floating point to ASCII conversion of the number in FPACC1. The ASCII string is placed in a buffer pointed to by the X index register. The buffer must be at least 14 bytes long to contain the ASCII conversion. The resulting ASCII string is terminated by a zero (0) byte. Upon exit the X index register will be pointing to the first character of the string. FPACC1 and FPACC2 will remain unchanged.

---

FLOATING POINT TO ASCII CONVERSION SUBROUTINE

FLTASC EQU *

SAVE THE POINTER TO THE STRING BUFFER.

FLTASC1 EQU *

SAVE THE POINTER TO THE STRING BUFFER.

FLTASC2 EQU *

SAVE THE POINTER TO THE STRING BUFFER.
LDAB FPACC1EX
SUBB #561
TAKEM BIAS +1.

LDAB FPACC1M+1
TAKEM BIAS -1.

ADD #23
ADDA
ADD IN THE NUMBER OF MANTISSA BITS -1.

BRA FLTASCM17
GO CHECK TO SEE IF WE NEED TO SHIFT AT ALL.

FLTASC7 LSR FPACC1MN
SHIFTMANTISSATOTHE RIGHT BY THE RESULT (MAKE

ROR FPACC1MN+1
THE NUMBER AN INTEGER).

ROR FPACC1MN+2

DECBL
DONE SHIFTING?

FLTASC7 BNE FLTASCM17
NO. KEEP GOING.

LDAA #1
GETINITIALVALUEOF"DIGITAFTERD.P."COUNT.

STAA 3,Y
INITIALIZE IT.

LDAA 2,Y
GETDECIMALEXPONENT.

ADDA #8
ADD THE NUMBER OF DECIMAL +1 TO THE EXPONENT.

* WAS THE ORIGINAL NUMBER > 9999999?

BMI FLTASC8
YES. MUST BE REPRESENTED IN SCIENTIFIC NOTATION.

BNE FLTASC8
YES. MUST BE REPRESENTED IN SCIENTIFIC NOTATION.

STAA 3,Y
MAKE THE DECIMAL EXPONENT THE DIGIT COUNT BEFORE

THEDECIMALPOINT.

LDAA #2
SET TO ZERO THE DECIMAL EXPONENT.

FLTASC8 SUBA #2
SUBTRACT2 FROM THE DECIMAL EXPONENT.

STAA 2,Y
SAVE THE DECIMAL EXPONENT.

TST 3,Y
DOES THE NUMBER HAVE AN INTEGER PART? (EXP. >0)

BGT FLTASC9
YES. GO PUT IT OUT.9

LDAA #0
NO. GET DECIMAL POINT.

LDAO 0,Y
GETPOINTERTOBUFFER.

STAA 0,X
PUTTHEDECIMALPOINTIN THE BUFFER.

INX POINT TO NEXT BUFFER LOCATION.

TST 3,Y
IS THE DIGIT COUNT TILL EXPONENT =0?

BEQ FLTASCM18
NO. NUMBER IS <1.

LDAA #0
YES. FORMAT NUMBER AS .XXXXXXX.

STAA 0,X
PUT THE 0 IN THE BUFFER.

INX POINT TO THE NEXT LOCATION.

FLTASC16 STX 0,Y
SAVE NEW POINTER VALUE.

FLTASC9 LDX #DECIMAL POINT TO THE TABLE OF DECIMAL DIGITS.

LDAO #7
INITIALIZE THE THE NUMBER OF DIGITS COUNT.

LDAA 18 A7 05

FLTASC16 CLR 4,Y
CLEAR THE DECIMAL DIGIT ACCUMULATOR.

FLTASC11 LD0 FPACC1MN+1
GETLOWER16 BitsOF MANTISSA.

SUBD 1,X
SUBTRACT LOWER 16 BITS OF CONSTANT.

STD FPACC1MN+1
SAVE RESULT.

LDAA FPACC1MN
GETUPPER 8BITS.

LDAA FPACC1MN
GETUPPER 8BITS.

SB0A,X
SUBTRACT UPPER 8BITS.

STAA FPACC1MN
SAVERESULT, DUE/LOWFLO?

BCS FLTASC12
YES. GO ADD DECIMAL NUMBER BACK IN.

INC 4,Y
ADD 1 TO DECIMAL NUMBER.

BRA FLTASCM11
TRY ANOTHER SUBTRACTION.

FLTASC12 LD0 FPACC1MN+1
GETFPACCl MANTISSALOW 16BITS.

ADDX 1,X
ADDLOW 16BITS BACK IN.

STD FPACC1MN+1
SAVE THE RESULT.

LDAA FPACC1MN
GETHIGH 8BITS.

ADDCA 0,X
ADD IN HIGH 8BITS OF CONSTANT.

STAA FPACC1MN
SAVE RESULT.

LDAA 4,Y
GETDIGIT.

ADDCA #30
MAKE IT ASCII.

LDX 0,Y
GET POINTER TO CONSTANTS.

STAA 0,X
PUTDIGITINBUFFER.

INX POINT TO NEXT BUFFER LOCATION.

DEC 3,Y
SHOULD WE PUT A DECIMAL POINT IN THE BUFFER YET?

BNE FLTASC16
NO. CONTINUE THECONVERSION.

LDAA #0.
YES. GET DECIMAL POINT.

LDAA 0,X
PUTIT IN THE BUFFER.

STAA 0,X
POINT TO THE NEXT BUFFER LOCATION.
0848 C403 CD EF 00  FLTASC16 STX 0,Y  SAVE UPDATED POINTER.
0849 C406 3B  PULX  RESTORE POINTER TO CONSTANTS.
0850 C407 0B  INX  POINT TO NEXT CONSTANT.
0851 C408 0B  INX
0852 C409 0B  INX
0853 C40A 18 6A 05  DEC 5,Y  DONE YET?
0854 C40D 26 8C  BNE FLTASC10  NO, CONTINUE CONVERSION OF "MANTISSA".
0855 C40F CD EE 00  LDX 0,Y  YES, POINT TO BUFFER STRING BUFFER.
0856 C402 09  FLTASC13 DEX  POINT TO LAST CHARACTER PUT IN THE BUFFER.
0857 C433 A6 00  LDAA 0,X  GET IT.
0858 C435 B1 30  CMPA #30  WAS IT an ASCII 0?
0859 C437 27 F9  BEQ FLTASC13  YES, REMOVE TRAILING ZEROS.
0860 C439 0B  INX  POINT TO NEXT AVAILABLE LOCATION IN BUFFER.
0861 C43A 18 E6 02  LDAB 2,Y  DO WE NEED TO PUT OUT an EXPONENT?
0862 C43D 27 2A  BEQ FLTASC15  NO, WE'RE DONE.
0863 C43F B6 45  LDAA #E  YES, PUT AN 'E' IN THE BUFFER.
0864 C441 A7 00  STAA 0,X
0865 C442 0B  INX  POINT TO NEXT BUFFER LOCATION.
0866 C444 B6 2B  LDAA #**  ASSUME EXPONENT IS POSITIVE.
0867 C446 A7 00  STAA 0,X  PUT PLUS SIGN IN THE BUFFER.
0868 C448 5D  TSTB  IS IT REALLY MINUS?
0869 C449 2A 05  BPL FLTASC14  NO, IS'S OK AS IS.
0870 C44B 50  NEGB  YES, MAKE IT POSITIVE.
0871 C44C B6 2D  LDAA #**  PUT THE MINUS SIGN IN THE BUFFER.
0872 C44E A7 00  STAA 0,X
0873 C500 0B  FLTASC14 INX  POINT TO NEXT BUFFER LOCATION.
0874 C501 CD EF 00  STX 0,Y  SAVE POINTER TO STRING BUFFER.
0875 C504 4F  CLRA  SET UP FOR DIVIDE.
0876 C505 CE 00 0A  LDX #10  DIVIDE DECIMAL EXPONENT BY 10.
0877 C50B 02  IDIV
0878 C509 37  PSHB  SAVE REMAINDER.
0879 C50A 8F  XGDX  PUT QUOTIENT IN D.
0880 C50B CB 30  ADDB #30  MAKE IT ASCII.
0881 C50D CD EE 00  LDX 0,Y  GET POINTER.
0882 C510 E7 00  STAB 0,X  PUT NUMBER IN BUFFER.
0883 C512 08  INX  POINT TO NEXT LOCATION.
0884 C513 33  PULB  GET SECOND DIGIT.
0885 C514 CB 30  ADDB #30  MAKE IT ASCII.
0886 C516 E7 00  STAB 0,X  PUT IT IN THE BUFFER.
0887 C518 08  INX  POINT TO NEXT LOCATION.
0888 C519 6F 00  FLTASC15 CLR 0,X  TERMINATE STRING WITH A ZERO BYTE.
0889 C51B 3B  PULX
0890 C51C 3B  PULX  CLEAR LOCALS FROM STACK.
0891 C51D 3B  PULX
0892 C51E 8D CB 43  JSR PULFPAC2  RESTORE FPACC2.
0893 C521 32  PULA
0894 C522 97 04  STAA MANTSGW1
0895 C524 38  PULX  RESTORE FPACC1.
0896 C525 DF 02  STX FPACC1M+1
0897 C527 38  PULX
0898 C528 DF 00  STX FPACC1E
0899 C52A 38  PULX  POINT TO THE START OF THE ASCII STRING.
0900 C52B 39  RTS  RETURN.
0901 *
0902 *
0903 C52C  DECOD1 EQU *
0904 C52C DF 42 40  FCB $0F,$42,$40  DECIMAL 1,000,000
0905 C52F 01 86 40  FCB $01,$86,$40  DECIMAL 100,000
0906 C532 00 27 10  FCB $00,$27,$10  DECIMAL 10,000
0907 C535 00 03 E8  FCB $00,$03,$E8  DECIMAL 1,000
0908 C538 00 00 64  FCB $00,$00,$64  DECIMAL 100
0909 C53B 00 00 0A  FCB $00,$00,$0A  DECIMAL 10
0910 C53E 00 00 01  FCB $00,$00,$01  DECIMAL 1
0911 *
0912 *
0913 C541  P9999999 EQU *
            CONSTANT 9999999.9

345
0914 C541 94 74 23 FE  FCB $94,$74,$23,$FE
0915
0916 C545  N999999 EQU *  CONSTANT 999999.
0917 C545 98 10 96 7F  FCB $98,$10,$96,$7F
0918
0919 C549  CONSTP5 EQU *  CONSTANT .5
0920 C549 80 00 00 00  FCB $80,$00,$00,$00
0921
0922
0923 C54D  FLTCHR EQU *
0924 C54D 7D 00 04  TST MANTSIGN1 IS FPACC1 NEGATIVE?
0925 C550 2A 12  BPL FLTCHR2 NO. CONTINUE WITH COMPARE.
0926 C552 7D 00 09  TST MANTSIGN2 IS FPACC2 NEGATIVE?
0927 C555 2A 00  BPL FLTCHR2 NO. CONTINUE WITH COMPARE.
0928 C557 DC 05  LDD FPACC2EX YES. BOTH ARE NEGATIVE SO COMPARE MUST BE DONE
0929 C559 1A 93 00  CPD FPACC1EX BACKWARDS. ARE THEY EQUAL SO FAR?
0930 C55C 26 05  BNE FLTCHR1 NO. RETURN WITH CONDITION CODES SET.
0931 C55E DC 07  LDD FPACC2MN+1 YES. COMPARE LOWER 16 BITS OF MANTISSAS.
0932 C560 1A 93 02  CPD FPACC1MN+1
0933 C563 39  FLTCHR1 RTS RETURN WITH CONDITION CODES SET.
0934 C564 96 04  FLTCHR2 LDAA MANTSIGN1 GET FPACC1 MANTISSA SIGN.
0935 C566 91 09  CMPA MANTSIGN2 BOTH POSITIVE?
0936 C568 26 F9  BNE FLTCHR1 NO. RETURN WITH CONDITION CODES SET.
0937 C56A DC 00  LDD FPACC1EX GET FPACC1 EXPONENT & UPPER 8 BITS OF MANTISSA.
0938 C56C 1A 93 05  CPD FPACC2EX SAME AS FPACC2?
0939 C56F 26 F2  BNE FLTCHR1 NO. RETURN WITH CONDITION CODES SET.
0940 C571 DC 02  LDD FPACC1MN+1 GET FPACC1 LOWER 16 BITS OF MANTISSA.
0941 C573 1A 93 07  CPD FPACC2MN+1 COMPARE WITH FPACC2 LOWER 16 BITS OF MANTISSA.
0942 C576 39  RTS RETURN WITH CONDITION CODES SET.
0943
0944
0945
This subroutine performs "unsigned" integer to floating point conversion of a 16 bit word. The 16 bit integer must be in the lower 16 bits of FPACC1 mantissa. The resulting floating point number is returned in FPACC1.

EQU LOX #FPACC1EX POIN T TO FPACC1.
JSR CHCKO IS IT ALREADY O?
BNE UINTFLT1 NO, GO CONVERT.
RTS YES, JUST RETURN.

LOA #$98 GET BIAS + NUMBER OF BITS IN MANTISSA.
STAA FPACC1EX INITIALIZE THE EXPONENT.
JSR FPNORM GO MAKE IT A NORMALIZED FLOATING POINT VALUE.
CLC NO ERRORS.
RTS RETURN.

SIGNED INTEGER TO FLOATING POINT

This routine works just like the unsigned integer to floating point routine except the 16 bit integer in the FPACC1 mantissa is considered to be in two's complement format. This will return a floating point number in the range -32768 to +32767.

EQU LOO FPACC1MN+1 GET THE LOWER 16 BITS OF FPACC1 MANT.
PSHA SAVE SIGN OF NUMBER.
BPL SINTFLT1 IF POSITIVE JUST GO CONVERT.
COMA MAKE POSITIVE.
COMB
ADDD #1 TWO'S COMPLEMENT.
STD FPACC1MN+1 PUT IT BACK IN FPACC1 MANTISSA.
SINTFLT1 BSR UINT2FLT GO CONVERT.
PULA GET SIGN OF ORIGINAL INTEGER.
LDAB #$FF GET "MINUS SIGN".
TSTA WAS THE NUMBER NEGATIVE?
BPL SINTFLT2 NO, RETURN.
STAB MANTSIGN YES, SET FPACC1 SIGN BYTE.
SINTFLT2 CLC NO ERRORS.
RTS RETURN.
FLOATING POINT TO INTEGER CONVERSION

This subroutine will perform "unsigned" floating point to integer conversion. The floating point number if positive, will be converted to an unsigned 16 bit integer (0 <= X <= 65535). If the number is negative it will be converted to a two's complement 16 bit integer. This type of conversion will allow 16 bit addresses to be represented as positive numbers when in floating point format. Any fractional number part is discarded.

FLOATING POINT TO INTEGER CONVERSION

Subroutine

FLT2INT EQU *

LDX #FPACCIEX PtOINT TO FPACC1.
JSR CHK0 IS IT 0?
BEQ FLT2INT3 YES, JUST RETURN.
LDA FLT2INT3 GET FPACC1 EXPONENT.
CMP #581 IS THERE AN INTEGER PART?
BLO FLT2INT 1 NO, GO PUT A 0 IN FPACC1.
TST MANTSIGN1 IS THE NUMBER NEGATIVE?
BMI FLT2INT1 YES, GO CONVERT NEGATIVE NUMBER.
CMP #590 IS THE NUMBER TOO LARGE TO BE MADE AN INTEGER?
BMI FLT2INT4 YES, RETURN WITH AN ERROR.
SUBB #59B SUBTRACT THE BIAS PLUS THE NUMBER OF BITS.
FLT2INT3 LSR FPACC1 MHN MAKE THE NUMBER AN INTEGER.
ROR FPACC1 MHN+1
ROR FPACC1 MHN+2
INC B DONE SHIFTING?
BNE FLT2INT5 NO, KEEP GOING.
CLR FPACC1 EX ZERO THE EXPONENT (ALSO CLEARS THE CARRY).
RTS

FLT2INT1 CMPB #5BF IS THE NUMBER TOO SMALL TO BE MADE AN INTEGER?
BMI FLT2INT4 YES, RETURN ERROR.
SUBB #59B SUBTRACT BIAS PLUS NUMBER OF BITS.
BSR FLT2INT5 GO DO SHIFT.
LDD FPACC1 MHN+1 GET RESULTING INTEGER.
COMA MAKE IT NEGATIVE.
COMB
ADD #1 TWO's COMPLEMENT.
STD FPACC1 MHN+1 SAVE RESULT.
CLR MANTSIGN1 CLEAR MANTISSA SIGN. (ALSO CLEARS THE CARRY)
RTS

FLT2INT4 LDA #TOLSgen NUMBER TOO LARGE OR TOO SMALL TO CONVERT TO INT.
SEC FLAG ERROR.
RTS

FLT2INT2 LDD #0

FLT2INT3 RTS RETURN.
This routine is used to calculate the square root of the floating point number in FPACC1. If the number in FPACC1 is negative an error is returned.

WORSE CASE = 16354 CYCLES = 8177 US @ 2MHz
LDD FPACCIEX GET FPACC1 EXPONENT & HIGH 8 BIT OF MANTISSA.
STD FPACCEX PUT IT IN FPACC2.
LDD FPACC1HH+1 GET FPACC1 LOW 16 BITS OF MANTISSA.
STD FPACC2HH+1 PUT IT IN FPACC2.
LDAA MANTSGN1 TRANSFER THE SIGN.
STAA MANTSGN2
RTS RETURN.
TTL FLTSIN

* FLOATING POINT SINE

* FLTSIN EQU *

JSR PSHFPAC2 SAVE FPACC2 ON THE STACK.
JSR ANGRED GO REDUCE THE ANGLE TO BETWEEN +/- PI.
PSHB SAVE THE QUAD COUNT.
PSHA SAVE THE SINE/COSINE FLAG.
JSR DEG2RAD CONVERT DEGREES TO RADIANS.
PULA RESTORE THE SINE/COSINE FLAG.
FLTSIN1 JSR SINCOS GO GET THE SINE OF THE ANGLE.
PULA RESTORE THE QUAD COUNT.
PULA RESTORE THE QUAD COUNT.
CMPA #2 WAS THE ANGLE IN QUADS 1 OR 2?
BLS FLTSIN2 YES. SIGN OF THE ANSWER IS OK.
COM MANTSGN1 NO. SINE IN QUADS 3 & 4 IS NEGATIVE.
FLTSIN2 CLC SHOW NO ERRORS.
JSR PULFPAC2 RESTORE FPACC2
RTS RETURN.
FLOATING POINT COSINE

**TTL FLTCOS**

* *

* FLOATING POINT COSINE *

* *

* *

FLTCOS EQU *

JSR PSHFPAC2 SAVE FPACC2 ON THE STACK.

JSR ANGRED GO REDUCE THE ANGLE TO BETWEEN +/-PI.

PSHB SAVE THE QUAD COUNT.

PSHA SAVE THE SINE/COSINE FLAG.

JSR DEG2RAD CONVERT TO RADIANS.

PULA RESTORE THE SINE/COSINE FLAG.

EORA #501 COMPLEMENT 90'S COMPLEMENT FLAG FOR COSINE.

JSR SINCO GO GET THE COSINE OF THE ANGLE.

PULA RESTORE THE QUAD COUNT.

CMPA #1 WAS THE ORIGINAL ANGLE IN QUAD 1?

BEQ FLTCOS1 YES. SIGN IS OK.

CMPA #4 WAS IT IN QUAD 4?

BEQ FLTCOS1 YES. SIGN IS OK.

COM MANTSIGN NO. COSINE IS NEGATIVE IN QUADS 2 & 3.

FLTCOS1 JMP FLTSIN2 FLAG NO ERRORS, RESTORE FPACC2, & RETURN.

* *

* *

*
SINCOS

**********************************************************************

SINCOS EQU *

PSHA SAVE SINE/COSINE FLAG ON STACK.

LDX FPACC1MN+1 SAVE THE VALUE OF THE ANGLE.

PSHX

LDX FPACC1EX JUST ALLOCATE ANOTHER LOCAL (VALUE NOT IMPORTANT)

PSHX GET INITIAL LOOP COUNT.

PSHA SAVE AS LOCAL ON STACK

LOX #$SINFACT POINT TO THE FACTORIAL TABLE.

PShX SAVE POINTER TO THE SINE FACTORIAL TABLE.

PSHX JUST ALLOCATE ANOTHER LOCAL (VALUE NOT IMPORTANT)

PSHA POINT TO LOCALS.

LDX #$COSFACT PUT THE NUMBER IN FW.

LDX #COSFACT GET THE NUMBER SIGN.

LDX #$COSFACT NO. GET POINTER TO COSINE FACTORIAL TABLE.

STX 1,Y SAVE IT.

JSR TR1TO2 TRANSFER FPACC1 TO FPACC2.

JSR FLTLMUL GET X'2' IN FPACC1.

JSR TF1TO2 COPY X'2' INTO FPACC2.

TST 10,Y ARE WE DOING THE SINE?

BEQ SINCOS7 YES. GO DO IT.

LDX #$COSFACT NO. GET POINTER TO COSINE FACTORIAL TABLE.

STX 1,Y SAVE IT.

JSR TR1TO2 TRANSFER FPACC1 TO FPACC2.

JSR TF1TO2 COPY X'2' INTO FPACC2.

LDX #$COSFACT NO. GET POINTER TO COSINE FACTORIAL TABLE.

LDAA #MANTSSN GET THE MANTISSA SIGN.

MANTSSN1 SAVE THE MANTISSA SIGN.

MANTSSN1 HAVE WE GENERATED ALL THE POWERS YET?

BNE SINCOS1 NO. DO MORE SOME MORE.

LDAA #$S4 SET UP LOOP COUNT.

STAA 0,Y GET NUMBER SIGN.

TSX 3,0 POINT TO POWERS ON THE STACK.

SINCOS2 SAVE THE POINTER.

SINCOS2 GET THE POINTER TO THE FACTORIAL CONSTANTS.

GETFPAC2 PUT THE NUMBER IN FPACC2.

INX POINT TO THE NEXT CONSTANT.

INX IT.

INX GET THE MANTISSA & EXPONENT.

FPACC1MN+1 PUT IT IN FPACC 1 EXPONENT & MANTISSA.

JSR FLTLMUL MULTIPLY THE TWO.

JSR FLTLMUL MULTIPLE THE TWO.

LDX 3,Y GET POINTER TO POWERS BACK.

LOD FPACC1MN+1 SAVE RESULT WHERE THE POWER OF X WAS.

STD 3,X GET Pointer to Powers Back.

STD 3,X

LOD FPACC1MN+1 SAVE RESULT WHERE THE POWER OF X WAS.

STD 1,X GET POINTER TO POWERS BACK.

STD 1,X

LOD FPACC1MN+1 SAVE RESULT WHERE THE POWER OF X WAS.

STD 1,X

LDAA MANTSSN1 SAVE SIGN.
1252 C702 A7 00  STAA 0,X  POINT TO THE NEXT POWER.
1253 C704 08  INX  
1254 C705 08  INX  
1255 C706 08  INX  
1256 C707 08  INX  
1257 C708 08  INX  
1258 C709 18 6A 00  DEC 0,Y  DONE?
1259 C70C 2D C5  BNE SINOS3 NO, GO DO ANOTHER MULTIPLICATION.
1260 C710 86 03  LOAA #3  GET LOOP COUNT.
1261 C710 1B A7 00  STAA 0,Y  SAVE IT.
1262 C713 CD EE 03  SINOS3  DEX 3,Y  POINT TO RESULTS ON THE STACK.
1263 C716 09  DEX  POINT TO PREVIOUS RESULT.
1264 C717 09  DEX  
1265 C718 09  DEX  
1266 C719 09  DEX  
1267 C71A 09  DEX  
1268 C71B CD EF 03  STX 3,Y  SAVE THE NEW POINTER.
1269 C71E A6 00  LOAA 0,X  GET NUMBERS SIGN.
1270 C720 97 09  STAA MANTSGN2 PUT IT IN FPAC2.
1271 C722 EC 01  LOD 1,X  GET LOW 16 BITS OF THE MANTISSA.
1272 C724 0D 05  STD FPAC2EX PUT IN FPAC2.
1273 C726 EC 03  LOD 3,Y  GET HIGH 8 BIT & EXPONENT.
1274 C728 0D 07  STD FPAC2MN+1 PUT IN FPAC2.
1275 C72A BD C2 3C  JSR FLTADD GO ADD THE TWO NUMBERS.
1276 C72C 18 6A 00  DEC 0,Y  DONE?
1277 C730 26 E1  BNE SINOS3 NO, GO ADD THE NEXT TERM IN.
1278 C732 1C 00 A0  TST 10,Y  ARE WE DOING THE SINE?
1279 C735 27 08  BEO SINOS5 YES, GO PUT THE ORIGINAL ANGLE INTO FPAC2.
1280 C737 CE C7 E3  LDX #ONE NO. FOR COSINE PUT THE CONSTANT 1 INTO FPAC2.
1281 C73A BD CB 66  JSR GETFPAC2  
1282 C73D 20 0F  BRA SINOS6 GO ADD IT TO THE SUM OF THE TERMS.
1283 C73F 18 A6 05  SINCOS5 LOAA 5,Y  GET THE VALUE OF THE ORIGINAL ANGLE.
1284 C742 97 09  STAA MANTSGN2 PUT IT IN FPAC2.
1285 C744 1E EC 06  LOD 6,Y  
1286 C747 DD 05  STD FPAC2EX  
1287 C749 1E EC 0B  LOD 8,Y  
1288 C74C 1F 07  STD FPAC2MN+1  
1289 C74F BD C2 3C  SINCOS6 JSR FLTADD GO ADD IT TO THE SUM OF THE TERMS.
1290 C751 30  TSX NOW CLEAN UP THE STACK.
1291 C752 8F  XGDX PUT STACK IN D.
1292 C753 C5 00 1F  ADDD #31 CLEAR ALL THE TERMS & TEMPS OFF THE STACK.
1293 C756 8F  XGDX  
1294 C757 35  TXS UPDATE THE STACK POINTER.
1295 C758 39  RTS RETURN.
1296 *  
1297 *  
1298 C759  ANGREO EQU  *  
1299 C75A 4F  CLRA INITIALIZE THE 45'S COMPLIMENT FLAG.
1300 C75A 36  PSHA PUT IT ON THE STACK.
1301 C75C 4C  INCA INITIALIZE THE QUAD COUNT TO 1.
1302 C75C 36  PSHA PUT IT ON THE STACK.
1303 C75D 18 30  TSY POINT TO IT.
1304 C75F CE C7 EB  LDX #THREE60 POINT TO THE CONSTANT 360.
1305 C762 BD CB 66  JSR GETFPAC2 GET IT INTO FPAC.
1306 C765 7D 00 04  TST MANTSGN1 IS THE INPUT ANGLE NEGATIVE:
1307 C768 2A 03  BPL ANGREO1 NO, SKIP THE ADD.
1308 C76A BD C2 3C  JSR FLTADD YES, MAKE THE ANGLE POSITIVE BY ADDING 360 DEG.
1309 C76D 7A 00 05  ANGREO1 DEC FPAC2EX MAKE THE CONSTANT IN FPAC2 90 DEGREES.
1310 C770 7A 00 05  DEC FPAC2EX  
1311 C773 BD C5 40  ANGREO2 JSR FLTOMP IS THE ANGLE LESS THAN 90 DEGREES ALREADY?
1312 C776 23 08  BLS ANGREO3 YES, RETURN WITH QUAD COUNT.
1313 C77B BD C2 FE  JSR FLTSSUB NO, REDUCE ANGLE BY 90 DEGREES.
1314 C77F 18 6C 00  INC 0,Y INCREASE THE QUAD COUNT.
1315 C77F 20 F3  BRA ANGREO2 GO SEE IF IT'S LESS THAN 90 NOW.
1316 C780 18 A6 00  ANGREO3 LOAA 0,Y GET THE QUAD COUNT.
1317 C785 B1 01  CMPA #1 WAS THE ORIGINAL ANGLE IN QUAD 1?
YES. COMPUTE TRIG FUNCTION AS IS.

NO. WAS THE ORIGINAL ANGLE IN QUAD 3?

YES. COMPUTE THE TRIG FUNCTION AS IF IN QUAD 1.

NO. MUST COMPUTE THE TRIG FUNCTION OF THE 90'S COMPLIMENT ANGLE.

ADD 90 DEGREES TO THE NEGATED ANGLE.

MAKE THE ANGLE IN FPACC 45 DEGREES.

IS THE ANGLE < 45 DEGREES?

YES. IT'S OK AS IT IS.

NO. MUST GET THE 90'S COMPLIMENT.

MAKE FPACC1 NEGATIVE.

GET THE 90'S COMPLIMENT.

SET THE FLAG.

GET THE QUAD COUNT.

GET THE QUAD COUNT & COMPLIMENT FLAG.

RETURN WITH THE QUAD COUNT & COMPLIMENT FLAG.

RETURN.

SINFACT EQU *

COSFACT EQU *

ONE FCB $81,$00,$00,$00

PI FCB $82,$49,$07,$08

THREE60 FCB $09,$34,$00,$00

2.0

3.1415927

5.0

10.0

100.0
HLT FLTTAN

* *
* FLOATING POINT TANGENT *
* *

* *

FLTAN EQU *

JSR PSHFPAC2 SAVE FPACC2 ON THE STACK.
JSR TFR1TOZ PUT A COPY OF THE ANGLE IN FPACC2.
JSR FLTCOS GET COSINE OF THE ANGLE.
JSR EXG1AND2 PUT RESULT IN FPACC2 & PUT ANGLE IN FPACC1.
JSR FLTSPIN GET SIN OF THE ANGLE.
JSR FLTDIV GET TANGENT OF ANGLE BY DOING SIN/COS.
BCC FLTTAN1 IF CARRY CLEAR, ANSWER OK.
LOX #MAXNUM TANGENT OF 90 WAS ATTEMPTED. PUT LARGEST
JSR GETFPAC1 NUMBER IN FPACC1.
LOA #TAN90ERR GET ERROR CODE IN A.
FLTTAN1 JSR PULFPAC2 RESTORE FPACC2.
RTS RETURN.

* *

MAXNUM EQU *

FCB $FE,$FF,$FF,LARGEST POSITIVE NUMBER WE CAN HAVE.

* *

356
The routines "DEG2RAD" and "RAD2DEG" are used to convert angles from degrees-to-radians and radians-to-degrees respectively. The routine "GETPI" will place the value of PI into FPPAC1. This routine should be used if the value of PI is needed in calculations since it is accurate to the full 24-bits of the mantissa.

```
1405 CB13
1406 CD16 CE CB 31
1407 CB19 BD CB 66
1408 CB1C BD C1 93
1409 CB1F BD CB 43
1410 CB22 39
1411 CB23
1412 CB26 CE CB 35
1413 CB29 20 EE
1414 CB31 CB35 86 65 2E E1
1415 CB31 7B 0E FA 35
1416 CE8 0E 93 35
1417 CB35 86 65 2E E1
1418 CB38
1419 CB39
1420 CB40
1421 CB41
1422 CB42
1423 CB43
1424 CB44
1425 CB45
1426 CB46
1427 CB47
1428 CB48
1429 CB49
1430 CB50
1431 CB51
1432 CB52
1433 CB53
1434 CB54
1435 CB55
1436 CB56
1437 CB57
1438 CB58
1439 CB59
1440 CB60
1441 CB61
1442 CB62
1443 CB63
```

```
DEG2RAD EQU *
JSR PSHPAC2 SAVE FPPAC2.
LDX #PIOV180 POINT TO CONVERSION CONSTANT PI/180.
DEG2RAD1 JSR GETPAC2 PUT IT INTO FPPAC2.
JSR FLIMUL CONVERT DEGREES TO RADIANS.
JSR PULPAC2 RESTORE FPPAC2.
RTS RETURN. (NOTE! DON'T REPLACE THE "JSR/RTS" WITH A "JMP" IT WILL NOT WORK.)

RAD2DEG EQU *
JSR PSHPAC2 SAVE FPPAC2.
LDX #C1800VPI POINT TO CONVERSION CONSTANT 180/PI.
BRA DEG2RAD1 GO DO CONVERSION & RETURN.

GETPI EQU *
LDX #PI POINT TO CONSTANT "PI".
JMP GETPAC1 PUT IT IN FPPAC1 AND RETURN.

PIOV180 EQU *
FCB $7B, $0E, $FA, $35

C1800VPI EQU *
FCB $86, $65, $2E, $E1
```
The following two subroutines, PSHFPAC2 & PULFPAC2, push FPACC2 onto and pull FPACC2 off of the hardware stack respectively. The number is stored in the "memory format".

PSHFPAC2 EQU * GET THE RETURN ADDRESS OFF OF THE STACK.
    PULX ALLOCATE FOUR BYTES OF STACK SPACE.
    PSHX PUT THE RETURN ADDRESS IN D.
    PSHX POINT TO THE STORAGE AREA.
    XGDX PUT THE RETURN ADDRESS BACK ON THE STACK.
    TSX
    PSHA
    JMP PULFPAC2
    PUTFPAC2 GO PUT FPACC2 ON THE STACK & RETURN.

PULFPAC2 EQU *
    TSX POINT TO THE RETURN ADDRESS.
    TSX POINT TO THE SAVED NUMBER.
    INX
    INX
    JSR GETFPAC2 RESTORE FPACC2.
    PULX GET THE RETURN ADDRESS OFF THE STACK.
    INX REMOVE THE NUMBER FROM THE STACK.
    INS INS
    INS
    JSR
    JMP 0,X RETURN.
**GETFPAC subroutine**

The GETFPAC1 and GETFPAC2 subroutines get a floating point number stored in memory and put it into either FPACC1 or FPACC2 in a format that is expected by all the floating point math routines. These routines may easily be replaced to convert any binary floating point format (e.g., IEEE format) to the format required by the math routines. The "memory" format converted by these routines is shown below:

```
exponent = mantissa
```

The exponent is biased by 128 to facilitate floating point comparisons. The sign bit is 0 for positive numbers and 1 for negative numbers. The mantissa is stored in hidden bit normalized format so that 24 bits of precision can be obtained. Since a normalized floating point number always has its most significant bit set, we can use the 24th bit to hold the mantissa sign. This allows us to get 24 bits of precision in the mantissa and store the entire number in just 4 bytes. The format required by the math routines uses a separate byte for the sign, therefore each floating point accumulator requires five bytes.

---

**GETFPAC1 subroutine**

```
LDD 0,X       GET THE EXPONENT & HIGH BYTE OF THE MANTISSA,
BEQ GETFP12 IF NUMBER IS ZERO, SKIP SETTING THE MS BIT.
CLR MANTSGN1 SET UP FOR POSITIVE NUMBER.
TSTB         IS NUMBER NEGATIVE?
BPL GETFP11 NO. LEAVE SIGN ALONE.
COM MANTSGN1 YES. SET SIGN TO NEGATIVE.
ORAS t/S80 RESTORE MOST SIGNIFICANT BIT IN MANTISSA.
STO FPACC1EX PUT IN FPACC1.
LOD 2,X       GET LOW 16-BITS OF THE MANTISSA.
STD FPACC1MN+1 PUT IN FPACC1.
RTS           RETURN.
```

**GETFPAC2 subroutine**

```
LDD 0,X       GET THE EXPONENT & HIGH BYTE OF THE MANTISSA,
BEQ GETFP22 IF NUMBER IS 0, SKIP SETTING THE MS BIT.
CLR MANTSGN2 SET UP FOR POSITIVE NUMBER.
TSTB         IS NUMBER NEGATIVE?
BPL GETFP21 NO. LEAVE SIGN ALONE.
COM MANTSGN2 YES. SET SIGN TO NEGATIVE.
ORAB #80     RESTORE MOST SIGNIFICANT BIT IN MANTISSA.
GETFP22 STD FPACC2EX PUT IN FPACC2.
LDD 2,X       GET LOW 16-BITS OF THE MANTISSA.
STD FPACC2MN+1 PUT IN FPACC2.
RTS           RETURN.
```
These two subroutines perform the opposite function of GETFPAC1 and GETFPAC2. Again, these routines are used to convert from the internal format used by the floating point package to a "memory" format. See the GETFPAC1 and GETFPAC2 documentation for a description of the "memory" format.

PUTFPAC1 EQU  *
LDD FPACC1EX GET FPAC1 EXPONENT & UPPER 8 BITS OF MANT.
TST MANTSGN1 IS THE NUMBER NEGATIVE?
BMI PUTFP11 YES. LEAVE THE M.S. BIT SET.
ANDB #$7F NO. CLEAR THE M.S. BIT.

PUTFP11 STD 0,X SAVE IT IN MEMORY
LDD FPACC1MN+1 GET L.S. 16 BITS OF THE MANTISSA.
STD 2,X
RTS  *

PUTFPAC2 EQU  *
LDD FPACC2EX GET FPAC2 EXPONENT & UPPER 8 BITS OF MANT.
TST MANTSGN2 IS THE NUMBER NEGATIVE?
BMI PUTFP21 YES. LEAVE THE M.S. BIT SET.
ANDB #$7F NO. CLEAR THE M.S. BIT.

PUTFP21 STD 0,X SAVE IT IN MEMORY
LDD FPACC2MN+1 GET L.S. 16 BITS OF THE MANTISSA.
STD 2,X
RTS  *
TFR1T02 C646 *1123 1091 1104 1209 1215 1383
THREE60 C7EB *1369 1304
TOLGSMER 0005 *0042 1049
UINT2FLT C577 *0959 0992
UINTFLT1 C580 *0964 0962
UMULT C109 *0395 0387
UMULT1 C1E3 *0402 0418
UMULT2 C1F4 *0411 0404
UMULT3 C217 *0426 0420
UMULT4 C234 *0439 0427 0434
UNFERR 0003 *0040 0382 0541 0664
Using the Serial Peripheral Interface to Communicate Between Multiple Microcomputers

As the complexity of user applications increases, many designers find themselves needing multiple microprocessors to provide necessary functionality in a circuit. Communication between multiple processors can often be difficult, especially when differing processors are used. A possible solution to this problem is usage of the serial peripheral interface (SPI), an interface intended for communication between integrated circuits on the same printed wire board. The MC68HC05C4 is one of the first single-chip microcomputers to incorporate SPI into hardware. One advantage of the SPI is that it can be provided in software, allowing communication between two microprocessors where one has SPI hardware and one does not. Special interfacing is necessary when using the hardware SPI to communicate with a microcomputer that does not include SPI hardware. This interface can be illustrated with a circuit used to display either temperature or time, that incorporates both a MC68HC05C4 and a MC68706R3. The MC68HC05C4 monitors inputs from a keypad and controls the SPI data exchange, while the MC68706R3 determines temperature by performing an analog-to-digital conversion on inputs from a temperature sensor and controls the LED display. Communication between the microcomputers is handled via SPI, with the MC68HC05C4 handling exchanges in hardware, and the MC68706R3 handling them in software.

Usage of software SPI can be expanded to include circuits where the single-chip implementing the SPI in software controls the data exchange, and those in which neither single-chip has hardware SPI capability. Minor modifications to the SPI code are necessary when data exchanges are controlled by the software.

Debugging designs including multiple processors can often be confusing. Some of the confusion can be alleviated by careful planning of both the physical debugging environment and the order in which software is checked.

SERIAL PERIPHERAL INTERFACE

Communication between the two processors is handled via the serial peripheral interface (SPI). Every SPI system consists of one master and one or more slaves, where a master is defined as the microcomputer that provides the SPI clock, and a slave is any integrated circuit that receives the SPI clock from the master. It is possible to have a system where more than one IC can be master, but there can only be one master at any given time. In this design, the MC68HC05C4 is the master and the MC68706R3 is the slave. Four basic signals, master-out/slave-in (MOSI), master-in/slave-out (MISO), serial clock (SCK), and slave select (SS), are needed for an SPI. These four signals are provided on the MC68HC05C4 on port D, pins 2-5.

SIGNALS

The MOSI pin is configured as a data output on the master and a data input on the slave. This pin is used to transfer data serially from the master to a slave, in this case the MC68HC05C4 to the MC68706R3. Data is transferred most significant bit first.

Data transfer from slave to master is carried out across the MISO, master-in/slave-out, line. The MISO pin is configured as an input on the master device and an output on the slave device. As with data transfers across the MOSI line, data is transmitted most significant bit first.

All data transfers are synchronized by the serial clock. One bit of data is transferred every clock pulse, and one byte can be exchanged in eight clock cycles. Since the serial clock is generated by the master, it is an input on the slave. The serial clock is derived from the master’s internal processor clock, and clock rate is selected by setting bits 0 and 1 of the serial peripheral control register to choose one of four divide-by values. Values for the MCUs crystal oscillators and the SPI divide-by must be chosen so that the SPI clock is no faster than the internal processor clock on the slave.

The last of the four SPI signals is the slave select (SS). Slave select is an active low signal, and the SS pin is a fixed input which is used to enable a slave to receive data. A master will become a slave when it detects a low level on its SS line. In this design, the MC68HC05C4 is always the master, so its SS line is tied to VDD through a pull-up resistor.

REGISTERS

Three registers unique to the serial peripheral interface provide control, status, and data storage.

The Serial Peripheral Control Register (SPCRI), shown below, provides control for the SPI.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10A</td>
<td>SPE</td>
<td>SPE</td>
<td>MSTR</td>
<td>CPOL</td>
<td>CPHA</td>
<td>SPI1</td>
<td>SPO</td>
</tr>
</tbody>
</table>

RESET 0 0 0 0 1 U U

SPIE — Serial Peripheral Interrupt Enable
0 = SPIF interrupts disabled
1 = SPI interrupt if SPIF = 1

SPE — Serial Peripheral System Enable
0 = SPI system off
1 = SPI system on

MSTR — Master Mode Select
0 = Slave mode
1 = Master mode

365
Data for the SPI is transmitted and received via the Serial Peripheral Data Register (SPDR). A data transfer is initiated by the master writing to its SPDR. If the master is sending data to a slave, it first loads the data into the SPDR and then transfers it to the slave. When reading data, the data bits are gathered in the SPDR and then the complete byte can be accessed by reading the SPDR.

**DEMONSTRATION BOARD DESCRIPTION**

A keypad input from the user is used to choose the output display function. The MC68HC05C4 monitors the keypad, decodes any valid inputs, and sends the data to the MC68706R3. If the user has requested a temperature display, the MC68706R3 sends a binary value of temperature in degrees fahrenheit to the MC68HC05C4, where the value is converted to a Celsius binary coded decimal value and returned to the MC68706R3 to be displayed. The LEDs are common anode displays and are driven directly off of port B on the MC68705R3. If the user desires the circuit to function as a real-time clock, a starting time must be entered and transmitted from the MC68HC05C4 to the MC68705R3. Once the clock has been initialized, the MC68705R3 updates the clock every minute. Clock values are stored in memory, and when the circuit is functioning as a thermometer, the values in memory are updated as required to maintain clock accuracy.

**USING THE A/D CONVERTOR TO MONITOR TEMPERATURE**

Temperature monitoring is performed by the Motorola MTS102 silicon temperature sensor and the LM358 Dual Low-Power Operational Amplifier, as shown in the schematic in

---

### Table 1. Serial Peripheral Rate Selection

<table>
<thead>
<tr>
<th>SPR1</th>
<th>SPR0</th>
<th>Internal Processor Clock Divide By</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>32</td>
</tr>
</tbody>
</table>

---

![Data Clock Timing Diagram](image)
Figure 2. Variations in the base-emitter voltage of the Motorola MT5102 silicon temperature sensor are monitored by the MC87805R3, which converts these analog inputs to equivalent digital values in degrees fahrenheit. The sensor voltage is buffered, inverted, and amplified by a dual differential amplifier before entering the A/D converter. An amplifier gain of 16 is used, resulting in 20-millivolt steps per degree fahrenheit. Using a $V_{CC}$ of 5 volts, the maximum differential amplifier output is 3.8 volts, resulting in a temperature sensing range from -40 degrees to +140 degrees fahrenheit.

The output from the differential amplifier is connected to the A/D converter on the MC88705R3. A block diagram of the successive approximation A/D converter is shown in Figure 3. Provision is made for four separate external inputs and four internal analog channels.

Two different registers associated with the converter control channel selection, initiate a conversion, and store the result of a completed conversion. Both the external and the internal input channels are chosen by setting the lower 3 bits of the A/D Control Register (ACR). The internal input channels are connected to the $V_{RH}/V_{RL}$ resistor chain and may be used for calibration purposes.

The converter operates continuously, requiring 30 machine cycles per conversion. Upon completion of a conversion, the digital value of the analog input is placed in the A/D result register (ARR) and the conversion complete flag, bit 7 of the ACR, is set. Another sample of the selected input is taken, and a new conversion is started.

Conversions are performed internally in hardware by a simple bisection algorithm. The D/A converter (DAC) is initially set to $80$, the midpoint of the available conversion range. This value is compared with the input value and, if the input value is larger, $80$ becomes the new minimum conversion value and the DAC is once again set to the midpoint of the conversion range, which is now $50$. If the input value is less than $80$, $80$ becomes the maximum conversion value and the DAC is set to the midpoint of the new conversion range, in this case $40$. This process is repeated until all eight bits of the conversion are determined.

Quantizing errors are reduced to $+1/2$ LSB, rather than $+0.1$ LSB, through usage of a built-in $1/2$ LSB offset. Ignoring errors, the transition between 00 and 01 will occur at $1/2$ LSB above the voltage reference low, and the transition between 0FE and 0FF will occur 1-1/2 LSBs below voltage reference high.

The A/D converter returns a value of $30$ when given an input of zero degrees fahrenheit, so $30$ must be subtracted from the result before converting to celcius. This offset must also be considered when calibrating the sensor. Calibration of the temperature sensor can be performed by adjusting the variable resistor to produce a display of $00$ after a piece of ice has been placed on the temperature sensor for approximately one minute. A 00 display results from a value of $50$ in the ARR, so the variable resistor should be adjusted until this value is reached.

**COMMUNICATION CONSIDERATIONS**

In this application, an SPI read or write is initiated via an interrupt from the MCU desiring to write data. When any of the three function keys, display temperature, set time, or display time, are pressed, the MC68HC05C4, as master, sends the MC87805R3 an interrupt on the MC87805R3's INT pin. The MC68HC05C4 writes the key value to its serial peripheral data register, thereby initiating the SPI. It then waits for the SPIF bit to go high and returns to scanning the keypad.

At the same time the MC88HC05C4 is writing to its SPDR, the MC87805R3 sets a bit counter to eight and waits for the first SCK from the MC88HC05C4. After each clock pulse, the MC87805R3 checks the status of the data bit, sets the carry bit equal to the data bit, and rotates the carry bit left into a result register. The bit counter is decremented, compared to zero, and if not zero, the MC87805R3 waits for the next clock pulse and repeats the cycle.

To ensure proper data transfers, the internal processor clock of the MC87805R3 must be sufficiently faster than the SPI clock of the MC88HC05C4 to allow the MC87805R3 time to complete this routine before the MC88HC05C4 can send another bit. This requires the user to first write the code to handle the software SPI, count machine cycles, and then choose MCU oscillator values that allow the additional machine cycles required in a software SPI to be completed before the master can send another clock pulse to the slave.

For example, consider the following piece of code for the MC87805R3, a slave receiving data from the master.

```
| DATA IN | PORTC pin 5 |
| SCK    | PORTC pin 4 |
```

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>LDA #00</td>
</tr>
<tr>
<td>5</td>
<td>STA BITCT</td>
</tr>
<tr>
<td>10</td>
<td>NXT BRSET 4,PORTC,*</td>
</tr>
<tr>
<td>10</td>
<td>BRSET 5,PORTC,STR</td>
</tr>
<tr>
<td>6</td>
<td>STR ROL RESULT</td>
</tr>
<tr>
<td>6</td>
<td>DEC BITCT</td>
</tr>
<tr>
<td>4</td>
<td>BNE NXT</td>
</tr>
</tbody>
</table>

Execution of this code requires 43 machine cycles. The maximum oscillator speed for an MC87805R3 is 1 MHz, requiring an SPI clock no greater than 1/43 MHz. One way of obtaining this rate for the SPI clock is to run the MC68HC05C4 at 0.5 MHz and choose a divide-by 32 to generate the SPI clock.

If the user has selected a temperature display, it is necessary for the MC87805R3, as a slave, to send data to the MC88HC05C4 master. When the MC87805R3 is ready to send data, it interrupts the MC68HC05C4 via the MC88HC05C4's IRQ line. The MC88HC05C4 then writes to its serial peripheral data register to initiate the transfer and shifts in data bits sent from the MC87805R3 until the SPIF bit goes high. While the MC88HC05C4 is writing to its SPDR, the MC87805R3 program is setting a bit counter to 8. When it detects a clock pulse on the SCK pin, the data register is rotated left one bit, placing the MSB in the carry. The MOSS pin is then set equal to the carry bit, the bit counter is decremented and, if it is greater than zero, the process is repeated.
Figure 2. Serial Peripheral Interface Demonstration Schematic
ADDITIONAL USES OF SPI

Many variations of this usage of the SPI are possible. The three possibilities are hardware SPI at both master and slave, software SPI at the master and hardware at the slave, and software SPI at both master and slave. Table 1 shows the various MCUs that have SPI implemented in hardware.

SPI is fairly straightforward in a circuit where both master and slave have hardware SPI capability. In this case, the MCUs are connected as shown in Figure 4. Figure 4a illustrates a single master system, and Figure 4b shows a system where either MCU can be system master. When both master and slave have SPI capability in hardware, data transfers can be handled full duplex. For a single master system, both master and slave write the data to be transferred to their respective serial peripheral data registers. A data transfer is initiated when the master writes to its serial peripheral data register. A slave device can shift data at a maximum rate equal to the CPU clock, so clock values must be chosen that allow the slave to transfer data at a rate equal to the master’s transfer rate. In a multiple master system, the master must pull the slave’s SS line low prior to writing to its serial peripheral data register and initiating the transfer.

PROGRAMMING A MASTER FOR SOFTWARE SPI

When the master in an SPI system does not have hardware SPI capabilities, the resulting system is quite different. An SPI system with a master providing the SPI in software is shown in Figure 5. This system only requires two lines between the microcomputers; data and clock. A slave select line can be added for use with multiple slaves. If operated with one data line, the SPI will function half-duplex only. Data is stored in a register, rotated left one bit at a time, and a port pin is set equal to the data bit. The master then provides the serial clock by toggling a different port pin. A bit counter must also be used to count the eight bits in the byte. Bit manipulation instructions are very useful for implementing SPI in software.

One possible software implementation for a write from the master to the slave is shown below.

```
<table>
<thead>
<tr>
<th>DATA OUT</th>
<th>PORTC pin 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCK</td>
<td>PORTC pin 1</td>
</tr>
<tr>
<td>DATA IN</td>
<td>PORTC pin 2</td>
</tr>
</tbody>
</table>

LDX #08
LDA DATA
RPT ROLA
BCS SET
BCLR 0,PORTC
BSET 1,PORTC
BCLR 1,PORTC
BCS SET
BNE RPT
SET BSET 0,PORTC
BNE RPT
BRA CLK
```

Full duplex operation requires a second data line. One port pin is then devoted to data-out and one to data-in. Data transfer from slave to master is accomplished immediately before the SCK pin is toggled. The state of the data-in pin is tested, and the carry is then set equal to the data-in pin. This value is then rotated in to a result register. The modified code is shown below.

```
<table>
<thead>
<tr>
<th>DATA OUT</th>
<th>PORTC pin 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCK</td>
<td>PORTC pin 1</td>
</tr>
<tr>
<td>DATA IN</td>
<td>PORTC pin 2</td>
</tr>
</tbody>
</table>

LDX #08
LDA DATA
BCLR 1,PORTC
ROTA ROLA
BCS SET
BCLR 0,PORTC
BSET 1,PORTC
BCLR 3,PORTC,CLK
ROTA DATAIN
DECO
BNE RPT
SET BSET 0,PORTC
BRA DATAIN
```

Figure 3. A/D Block Diagram
SLAVE PERIPHERAL PROCESSOR MAY BE ANOTHER MC68HC04C4, OR PERHAPS AN M6804 FAMILY SINGLE CHIP MCU

Figure 4a. Single Master SPI

OTHER PROCESSORS MAY BE USED. SOME FORTHCOMING M6801, M6805, AND M6804 FAMILY VERSIONS HAVE SPI FACILITIES.

Figure 4b. Multiple Master SPI

Figure 5. Software SPI
PROGRAMMING A SLAVE FOR SOFTWARE SPI

If the slave in the system is a MCU with hardware SPI capability, the data transfer will happen automatically, one bit per clock pulse. If the slave is a MCU that does not have SPI implemented in hardware, a read requires the following actions. A bit counter is set to eight, the slave polls its SCK pin waiting for a clock transition, once it perceives a clock it checks its data-in pin, sets the carry equal to the data and rotates the carry into a results register. One possible code implementation is shown in the previous timing example.

Converting this to full duplex operation requires the addition of a write from slave to master. The slave rolls a data register to place the data bit to be sent into the carry, and the data-out pin is set equal to the carry. These actions occur prior to the read of data from the master. With these modifications, the code looks as shown below.

| DATA OUT | PORTC pin 6 |
| DATA IN  | PORTC pin 5 |
| SCK      | PORTC pin 4 |
| LDA      | #08        |
| STA      | BITCT      |
| AGN      | BRSET 4,PORTC,* |
| ROL      | RES1       |
| BCS      | SET1       |
| BCLR     | 6,PORTC    |
| BRCLR    | 4,PORTC,*  |
| BRSET    | 5,PORTC,STR|
| STR      | RESULT     |
| DEC      | BITCT      |
| BNE      | AGN        |

DEBUGGING TIPS

Debugging a circuit containing two microcomputers presents various problems not evident when working with a single microcomputer circuit. The first problem is simultaneously providing emulation for both microcomputers. Once emulation capability is arranged, the designer needs to keep track of the progress of each single-chip, and monitor how the actions of one affects the actions of the other.

One of the easiest methods to debug a circuit of this type is to use two emulator stations, complete with separate terminals. Any emulators can be used, but user confusion is reduced if the emulators have similar commands and syntax. Physical separation also helps reduce confusion. It is somewhat easier to keep track of the concurrent operations if one side of the prototype board is devoted to each single-chip and the majority of peripherals they each must interface with, and the emulator for that microcomputer is placed to that side of the printed circuit board.

Before starting simultaneous debugging, it is best to individually debug the code for each microcomputer wherever possible. Once it becomes necessary for the microcomputers to communicate with one another, halt one of the microcomputers anytime they are not actually talking and work with the remaining microcomputer. As the debugging progresses, keep in mind that an error in the function of one single-chip does not necessarily indicate an error in the corresponding code for that single-chip, but rather, the error may have been caused by an incorrect or unintended transmission from the other single-chip.

Although the aforementioned suggestions reduce debugging problems, some will remain. Long periods of debug can result in an obscuring of the separation of the functions of the two programs. It helps to take periodic breaks to get away from the system and clear the thought processes. Expect to occasionally be confused, be willing to retrace sections of code multiple numbers of times, and the debugging will proceed fairly smoothly.

CONCLUSION

The Serial Peripheral Interface can be used as a tool to innerconnect to MCU with various other MCUs or peripherals, and can be used with any microcomputer. A special case occurs when one, or more, of the MCUs in a circuit do not have SPI capability in hardware. In this case, a simple software routine can be written to perform the SPI. Used in this manner, the SPI eliminates the need for costly, inconvenient parallel expansion buses and Universal Asynchronous Receiver/Transmitters (UARTs) and simplifies the design effort.
**Register Address Definition**

- `porta` equ 0
- `portc` equ 2
- `portd` equ 3
- `ddra` equ 4
- `ddrc` equ 6
- `spsr` equ $0b
- `spcr` equ $0a
- `spdr` equ $0c
- `tcr` equ $12

**Keypad Lookup Table**

- `kypad` equ *
- `fcb` $07
- `fcb` $04
- `fcb` $01
- `fcb` $00
- `fcb` $08
- `fcb` $05
- `fcb` $02
- `fcb` $0a
- `fcb` $09
- `fcb` $06
- `fcb` $03
- `fcb` $0d
- `fcb` $0c
- `fcb` $0f
- `fcb` $0b

**Program Start**

- `start` rsp
- `clr` tcr
- `mask` timer
- `interruptions`
0064 0065
0066 0114 CO 01 67
0067 0117 A1 0A
0068 0119 27 OE
0069 011B A1 0E
0070 011D 27 2B
0071 011F A1 0F
0072 0121 27 06
0073 0123 A1 0B
0074 0125 27 02
0075 0127 20 0E
0076 0077
0078 0079 0129 11 02
0080 012B CO 01 59
0081 012E A1 0A
0082 0130 27 02
0083 0132 20 0E
0084 0085 0134 9A
0086 0135 20 0D
0087 0088
0089 0137 CO 01 67
0090 013A A1 0A
0091 013C 27 F9
0092 013E A1 0B
0093 0140 27 F5
0094 0142 A1 0E
0095 0144 27 F1
0096 0146 A1 0F
0097 0148 27 ED
0098 014A 11 02
0099 014C CO 01 59
0100 014F A1 0C
0101 0151 27 C1
0102 0153 A1 0D
0103 0155 27 BD
0104 0157 20 0E
0105 0106
0107 0108 0159
0109 0159 B7 0C
0110 015B 0F 0B FD
0111 015E 10 02
0112 0160 81
0113
0114 0115
0116 0161
0117 0161 BF 0C
0118 0163 0F 0B FD
0119 0166 81
0120

** check keypad **

key jsr keypad
cmp #$0a check for disp. temp
beq dtmp
cmp #$0e check for set time
beq sttm
cmp #$0f check for disp. time
beq dtmp
cmp #$0b check for disp. sec
beq dtmp
bra key wait for next input

** display temp **

dtmp bclr O.portc send interrupt for spi
jsr spiwr send byte
cmp #$0a check for disp. temp.
beq ciri
bra key

clr cli:
bra key

** set time **

nudig jsr keypad
cmp #$0a
beq nudig
 cmp #$0b
beq nudig look for valid digit
 cmp #$0e
beq nudig
 cmp #$0f
beq nudig

sttm bclr O.portc send int. for spi
jsr spiwr send value
 cmp #$0c check for pm
beq key yes, wait for next input
 cmp #$0d check for am
beq key yes, wait for next input
bra nudig get next time digit

** spi write subroutine **

spiwr equ *
st spdr put data in data reg.
brclr 7;spsr,* wait for end of byte
bset O.portc
spifig rts done

** spi read subroutine **

spird equ *
stx spdr initiate transfer
brclr 7;spsr,* wait for end of byte
rdend rts

373
** keypad scanning routine **

keypad equ *

** 32 msec delay **

wtlp lda #$20 set up outer loop
sta ct1 counter

otlp lda #$32 set up inner loop

inlp deca dec. inner loop
bne inlp when 0,
dec ct1 decrement outer loop
bne otlp

clrx set up row counter

lda #$80 check first row
sta porta

nxtr lda porta check for key
and #$0f mask upper nibble
cmp #$00 look for zero
bne debnc branch if have a key

asr porta try next row
incx decrement row counter
cpx #$03 check for zero

bis nxtr test next row

inval bra wtlp no key pressed

** debounce key input **

debnc sta tmpa save value

stx rwno save row number

lda #$ff set up delay

loop deca

bne loop wait

lda porta check row again
and #$0f mask upper nibble

cmp tmpa check for same key

bne wtlp return if invalid

** wait for key release **

wtr lda porta check value

and #$0f mask upper nibble
cmp #$00 look for zero

bne wtr wait for release

** decode key value **

lda tmpa restore value

cirx set up column ctr.

nxtc ilsra shift columns

bcs col branch if have column

incx

bra nxtc try next column

col ils lx x=4*col. no.

tsx place x in a

add rwno key value =4*col + row
tax place a in x

lda kypd,x convert to decimal
sta tmpa

rts
**** temperature conversion routine ****

*  fahrenheit value is received from 705r3 via
*  spi and received in the a register, the value
*  is converted to celcius and the leftmost
*  led is blanked.

r3int  equ *
   jsr  spird  read value
   lda  spcx  transfer value to register
   sub  #$20  subtract 32
   bhs  conv  if pos. convert
   nega  negate
   idx  #$0a  low
   stx  base+2  '-' pattern

**** temperature conversion ****
** a 16-bit multiply by 5 is performed on the
** value received from the r3, this number
** is then divided by 9.

conv  clr  msb  clear counters
       clr  lsb
       sta  lsb
       lsla  multiply by 2
       lsla  multiply by 2
       rol  msb  load overflow into msb
       rol  msb  load overflow into msb
       add  lsb  a contains value x5
       bcc  div  if overflow, inc msb
   div  clr  dctr
       sta  lsb
       inc  dctr
       sta  lsb
       ida  msb
       sbc  #$00  subtract borrow from msb
       sta  lsb
       ida  lsb  count factors of 9
       sub  #$09
       bcc  nxt9  if no borrow, repeat
       tst  msb  if borrow, check for end
       bne  nxt9  repeat if not end

*** end of divide, add last 9 back in and
*** check remainder for rounding

   add  #$09  find remainder
   cmp  #$04  if greater
   bhi  done  than 4, round up
   dec  dctr
   lda  dctr
   bcc  nxt9  if no borrow, repeat
   tst  msb  if borrow, check for end
   bne  nxt9  repeat if not end

pos  idx  #$0b  blank pattern
    stx  base+3  blank most sig. digit

**** convert binary value to bcd value ****

*  the x registers begins with the binary value
*  and exits with zero, each digit, units, tens
D245
D246
D247
D248
D249
D250
D251
D252
D253
D254
D255
D256
D257
D258
D259
D260
D261
D262
D263
D264
D265
D266
D267
D268
D269
D270
D271
D272
D273
D274
D275
D276
D277
D278
D279
D279
D280
D281
D282
D283
D284
D285
D286
D287
D288
D289
D290
D291
D292
D293
D294
D295

* and hundreds, is stored separately and checked
* for a value equal to 10.

D246
D247
D248
D249
D250
D251
D252
D253
D254
D255
D256
D257
D258
D259
D260
D261
D262
D263
D264
D265
D266
D267
D268
D269
D270
D271
D272
D273
D274
D275
D276
D277
D278
D279
D280
D281
D282
D283
D284
D285
D286
D287
D288
D289
D290
D291
D292
D293
D294
D295

* send all digits to 705r3 via spi
* start by interrupting r3 and then
* sequentially sending four values

D246
D247
D248
D249
D250
D251
D252
D253
D254
D255
D256
D257
D258
D259
D260
D261
D262
D263
D264
D265
D266
D267
D268
D269
D270
D271
D272
D273
D274
D275
D276
D277
D278
D279
D280
D281
D282
D283
D284
D285
D286
D287
D288
D289
D290
D291
D292
D293
D294
D295

*** initialize interrupt vectors ***

D246
D247
D248
D249
D250
D251
D252
D253
D254
D255
D256
D257
D258
D259
D260
D261
D262
D263
D264
D265
D266
D267
D268
D269
D270
D271
D272
D273
D274
D275
D276
D277
D278
D279
D280
D281
D282
D283
D284
D285
D286
D287
D288
D289
D290
D291
D292
D293
D294
D295

org $1ff4

spivec fdb start
scivec fdb start
tmvvec fdb start
irqvec fdb r3int
swivec fdb start
reset fdb start
**REGISTER DEFINITION**

- **portb** equ 1
- **portc** equ 2
- **ddrb** equ 5
- **ddrc** equ 6
- **tdr** equ 8
- **tcr** equ 9
- **acr** equ 14
- **arr** equ 15

```
0001 nam r3disp
********** REGISTER DEFINITION **********

0006 0001 portb equ 1
0007 0002 portc equ 2
0008 0005 ddrb equ 5
0009 0006 ddrc equ 6
0010 0008 tdr equ 8
0011 0009 tcr equ 9
0012 000E acr equ 14
0013 000F arr equ 15

org $40

0019 0040 wrdat rmb 1
0020 0041 timtmp rmb 1
0021 0042 ct rmb 1
0022 0043 cont rmb 1
0023 0044 result rmb 1
0024 0045 res1 rmb 1
0025 0046 bitct rmb 1
0026 0047 sec rmb 1
0027 0048 segmnt rmb 1
0028 0049 pm rmb 1
0029 004A base rmb 4

org $80

**** display look-up table ****

```
0090 start equ *

segtab equ *
  fcb %00000001 0
  fcb %01001111 1
  fcb %00010010 2
  fcb %00000110 3
  fcb %01001100 4
  fcb %00100100 5
  fcb %00000111 6
  fcb %00001111 7
  fcb %01111111 blank
  fcb %01111111 pm
  fcb %00011000 p

org $90 program start

*** initialize variables ***

```
0051 0090 start equ *

ida #$07
sta $f38 set MOR
ida #$ff
sta ddrb set up port b as output
sta tdr set timer for prescale of 128
sta base
sta base+1 blank time display
```

377
sta base+2
sta base+3
ida #$ef
sta portc
sta segmnt
ida #$cf
sta drc
ida #$0f
sta tcr
clr timtmp
clr sec
clr pm
ida #$3b
sta ct
ida #$08
sta ct1
cli

* delay for
dlay
ldx #$ff
ida #$ff
deca
bne dlay+4
decx
bne dlay+2

* temperature measurement *
clr acr
ida acr
bpl #$-2
ida arr
get result
sbc #$30
sta res1
store in spi data register
ida timtmp
cmp #$07
check for temp. update
bne dlay

*** send temperature value to hc05c4 for
* conversion into celcius. start by
* interrupting the hc05c4 and then transmit
* data via the spi.
sei
bclr 7, portc interrupt hc05c4
jsr spiw r
write data to hc05c4
ida #$04

* wait for return data ~ 140 cycles *

1012
1013
1014
1015
1016
1017 00DC 9B
1018 00DD 1F 02
1019 00DF CD 01 18
1010 00E2 AE 04
110
1112
1113
1114 00E4 A6 0B
1115 00E6 B7 51
1116 00EB A6 0E
1117 00EA 4A
1118 00EB 26 FD
1119
1120
1121
1122
1123 00ED CD 01 01
1124 00F0 B6 44

set portc to choose msd
set up c0-3,c6,c7 as outputs
unmask timer interrupt
start with time disp.
set seconds to zero
start with am
set up timing loops

sta base+7
sta base+7
ida #$0b
sta base+7
ida #$0e
timlp
deca
bne timlp

* get decimal values in celcius from
* hc05c4

ida result
get value

378
sta base,x
inc
cpx #$07
bne nxtdg
cli
bra delay

**
select temperature display**

lda #$07
sta timtmp
rti

******
spi routines******
* the three pins used for the spi are
* miso bit 6, portc
* mosi bit 5, portc
* sck bit 4, portc
* the r3 waits for a high-to-low
* transition of the spi clock, which
* is provided by the hc05c4 and sent
* on portc pin 4. a bit of data is
* transferred on each high-to-low
* transition of the clock.
* spi read *
lda #$08
sta bitct
set bit counter
brset 4, portc,* wait for clock transition
brset 5, portc,str check data status

* note: the brset command automatically
* sets the carry bit to be equal to the
* bit under test

rol result
ida #$02
delay loop

sta
decca
bne stall
nap
dec bitct
check for end of byte
bne nxt
get next bit

rts

# spi write *
lda #08
sta bitct
set bit counter
rol res1
shift data
bcs set1
check data status
bclr 6, portc
if 0, clear miso
brset 7, portc
clear interrupt
bclr 4, portc
nap

nnap
timing delay.

brset 4, portc,* wait for clock trans.
** initialization of data read via spi **

* a data read is initiated via an interrupt
  from the HC05C4. The value received is tested to determine which function is requested and the processor jumps to the proper routine.

** check for valid input **

* valid input, set timer counter *

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0187 012B 3A 46</td>
<td>dec bitct, check for end of byte</td>
</tr>
<tr>
<td>0188 012D 26 E0</td>
<td>bne agn, clear interrupt</td>
</tr>
<tr>
<td>0189 012F 1E 02</td>
<td>bset 7,portc, rts</td>
</tr>
<tr>
<td>0190 0131 B1</td>
<td>set1, bset 6,portc, if 1, set miso</td>
</tr>
<tr>
<td>0191</td>
<td>bset 7,portc, clear interrupt</td>
</tr>
<tr>
<td>0192 0132 1C 02</td>
<td>bset 4,portc, bra tst</td>
</tr>
<tr>
<td>0193 0134 1E 02</td>
<td>c4int, jsr spird, set value</td>
</tr>
<tr>
<td>0194 0136 18</td>
<td>ldi, lda #$03, sta timtmp, choose time</td>
</tr>
<tr>
<td>0195 0138 20</td>
<td>lda result, cmp #$0a, check for disp temp</td>
</tr>
<tr>
<td>0196</td>
<td>beq temp, cmp #$0f, check for display time</td>
</tr>
<tr>
<td>0197</td>
<td>beq rtry, cmp #$0e, check for set time</td>
</tr>
<tr>
<td>0198</td>
<td>beq clrtm, cmp #$0d, check for am</td>
</tr>
<tr>
<td>0199</td>
<td>beq am, cmp #$0b, check for secs</td>
</tr>
<tr>
<td>0200</td>
<td>beq dsec, cmp #$0c, check for pm</td>
</tr>
<tr>
<td>0201</td>
<td>bne dig, cmp #$1f, set pm address</td>
</tr>
<tr>
<td>0202</td>
<td>sta pm, sta base+3, check tens of hours</td>
</tr>
<tr>
<td>0203</td>
<td>beq blhr, if zero, blank digit</td>
</tr>
<tr>
<td>0204</td>
<td>cmp #$01, beq twoc</td>
</tr>
<tr>
<td>0205</td>
<td>cmp #$0b, look for blank</td>
</tr>
<tr>
<td>0206 013A CD 01 01</td>
<td>bne blank, if not, blank display</td>
</tr>
<tr>
<td>0207 013D A6 03</td>
<td>blhr, lda #$0b, sta base+3, blank tens of hours</td>
</tr>
<tr>
<td>0208 013F B7 41</td>
<td>lda base+1, check tens of minutes</td>
</tr>
<tr>
<td>0209 0141 B6 44</td>
<td>cmp #$05, check against 5</td>
</tr>
<tr>
<td>0210 0143 A1 0A</td>
<td>bhi blank, if greater, blank display</td>
</tr>
<tr>
<td>0211 0145 27 B5</td>
<td>lda #$1f, unmask timer interrupt</td>
</tr>
<tr>
<td>0212 0147 1A 0E</td>
<td>sta tcr, lda #$43, load inner loop counter</td>
</tr>
<tr>
<td>0213 0149 39</td>
<td>lda #$06, sta ct, load outer loop counter</td>
</tr>
<tr>
<td>0214 014F 27</td>
<td>lda #00, cmp #$0a, check against 5</td>
</tr>
<tr>
<td>0215 0151 39</td>
<td>beq clrtm, cmp #$0d, check for am</td>
</tr>
<tr>
<td>0216 0153 27 C0</td>
<td>beq am, cmp #$0b, check for secs</td>
</tr>
<tr>
<td>0217 0155 6A</td>
<td>beq dsec, cmp #$0c, check for pm</td>
</tr>
<tr>
<td>0218 0157 01 0C</td>
<td>bne dig, cmp #$1f, set pm address</td>
</tr>
<tr>
<td>0219 0159 27 39</td>
<td>sta pm, lda base+3, check tens of hours</td>
</tr>
<tr>
<td>0220 015B A6 FF</td>
<td>beq blhr, if zero, blank digit</td>
</tr>
<tr>
<td>0221 015D B7 49</td>
<td>cmp #$01, beq twoc</td>
</tr>
<tr>
<td>0222 015F B6 4D</td>
<td>cmp #$0b, look for blank</td>
</tr>
<tr>
<td>0223</td>
<td>bne blank, if not, blank display</td>
</tr>
<tr>
<td>0224</td>
<td>lda #$0b, sta base+3, blank tens of hours</td>
</tr>
<tr>
<td>0225</td>
<td>lda base+1, check tens of minutes</td>
</tr>
<tr>
<td>0226</td>
<td>cmp #$05, check against 5</td>
</tr>
<tr>
<td>0227 015F B6 4D</td>
<td>bhi blank, if greater, blank display</td>
</tr>
<tr>
<td>0228 0161 27 08</td>
<td>lda +%3, unmask timer interrupt</td>
</tr>
<tr>
<td>0229 0163 A1 01</td>
<td>sta tcr, lda #$43, load inner loop counter</td>
</tr>
<tr>
<td>0230 0165 27 46</td>
<td>lda #$06, sta ct, load outer loop counter</td>
</tr>
<tr>
<td>0231 0167 A1 0B</td>
<td>lda +%3, check tens of hours</td>
</tr>
<tr>
<td>0232 0169 26 48</td>
<td>lda #00, cmp #$0a, check against 5</td>
</tr>
<tr>
<td>0233 016B A6 08</td>
<td>beq blhr, if zero, blank digit</td>
</tr>
<tr>
<td>0234 016D B7 4D</td>
<td>cmp #$01, beq twoc</td>
</tr>
<tr>
<td>0235 016F B6 4B</td>
<td>cmp #$0b, look for blank</td>
</tr>
<tr>
<td>0236 0171 A1 05</td>
<td>bne blank, if not, blank display</td>
</tr>
<tr>
<td>0237 0173 22 3E</td>
<td>lda +%3, unmask timer interrupt</td>
</tr>
<tr>
<td>0238</td>
<td>lda $1f, sta tcr, load inner loop counter</td>
</tr>
<tr>
<td>0239</td>
<td>lda $43, lda #$06, sta ct, load outer loop counter</td>
</tr>
<tr>
<td>0240</td>
<td>lda $1f, sta tcr, load inner loop counter</td>
</tr>
<tr>
<td>0241 0175 A6 0F</td>
<td>lda +%3, unmask timer interrupt</td>
</tr>
<tr>
<td>0242 0177 B7 09</td>
<td>lda $1f, sta tcr, load inner loop counter</td>
</tr>
<tr>
<td>0243 0179 A6 43</td>
<td>lda $43, lda #$06, sta ct, load outer loop counter</td>
</tr>
<tr>
<td>0244 017B B7 42</td>
<td>lda %3, cmp #$05, check against 5</td>
</tr>
<tr>
<td>0245 017D A6 06</td>
<td>lda +%3, unmask timer interrupt</td>
</tr>
<tr>
<td>0246 017F B7 43</td>
<td>lda #1f, cmp #$0a, check against 5</td>
</tr>
<tr>
<td>0247 0181 3F 47</td>
<td>beq blhr, if zero, blank digit</td>
</tr>
<tr>
<td>0248 0183 3F 52</td>
<td>cmp #$0b, lda +%3, unmask timer interrupt</td>
</tr>
</tbody>
</table>
**clear displays**

clr base+9

rtry rti

* input digit *

*** time setting routine ***

time is inputted left to right

and the end of input is indicated

be pressing either the am or pm

button. pm is denoted on the

display by lighting the decimal

point. counters are set to zero out

after each second.

**check if time less than 12 o'clock**

blank display if not

**seconds display**

blank first two leds

set timtmp to 0b

blank 1st two leds

rti
0311
0312
0313
0314
0315
0316
0317
0318
0319
0320
0321 01CB
0322
0323 01CB 99
0324 01C9 BE 41
0325 01CB A6 FF
0326 01CD B7 01
0327 01CF 36 48
0328 01D1 25 04
0329
0330
0331 01D3 A6 F7
0332 01D5 B7 48
0333 01D7 E6 4A
0334 01D9 5A
0335 01DA 02 48 02
0336 01DD E6 4A
0337 01DF 5A
0338 01E0 04 48 02
0339 01E3 E6 4A
0340 01E5 5A
0341 01E6 06 48 02
0342 01E9 E6 4A
0343 01EB A4 0F
0344 01ED 97
0345 01EE EE 80
0346 01F0 BF 01
0347 01F2 B6 48
0348 01F4 B7 02
0349
0350
0351
0352
0353
0354
0355 01F6 A6 10
0356 01FB B7 08
0357 01FA A6 0F
0358 01FC B7 09
0359 01FE 3A 42
0360 0200 26 08
0361 0202 A6 3B
0362 0204 B7 42
0363 0206 3A 43
0364 0208 27 01
0365 020A 80
0366
0367
0368
0369
0370
0371
0372

***** display routine *****
*
* displays are refreshed every msec
* when a timer interrupt occurs. the
* most significant digit is displayed
* first. at the conclusion of each
* minute, the time is updated
*
* tmrint equ *

sec
ldx timtmp choose time or temp
lda #$f7 restart with msd
sta segmnt

min2 lda base,x load a with minutes
decx point to next digit
brset 1,segmnt,hrs1 check hours units
lda base,x load a with tens of min.
hrs1 decx point to next digit
brset 2,segmnt,hrs2 check tens of hrs.
lda base,x load a with hours units
hrs2 decx point to next digit
brset 3,segmnt,disp display value
disp and #$0f mask upper nibble
tax set x equal to a
ldx segtab,x display value table
stx portb enable display drivers
lda segmnt
sta portc enable display

** count display refreshes. 402 refreshes
* equals one second. after 402 refreshes,
* update clock
*
ida #$10 set timer to interrupt
sta tdr after 2048 cycles
ida #$0f sta tcr reset timer interrupt flag
dec ct decrement inner loop
bne ret
ida #$3b sta ct reset inner loop
dec ctt1 decrement outer loop
beq tmcfg if one sec., to time change
ret rti

***** time change routine *****
*
* when 60 seconds are counted;
* increase minutes by one; if
* necessary, blank minutes and increase
* hours. change am/pm if needed.
*
382
tms## equ *
inc sec
inc base+8
lda base+8
cmp #$0a
beq tens
lda #$3d
inc sec
bne ret1
wait for next second
cmp sec
clr sec
cmp base+9
da base
inc
check min. units = 0
cmp #$07
 lda base+1
inc
check tens of min.
cmp #$05
less than 5?
 lda base+1
inc
check tens of min.
cmp #$09
less than 9?
 lda base
inc
check min. units
beq hrck
lda base+3
inc
check tens of hrs.
cmp #$0b
hrck
lda base+2
inc
check hrs. units
cmp #$02
hrck
lda base+1
inc
check tens of min.
cmp #$08
hrck
lda base+2
inc
tens of min = 0
cmp base+1
da base+3
inc
check tens of hrs.
cmp base+3
da base+2
inc
check tens of min.
cmp base+2
da base+1
inc
check tens of hrs.
cmp base+1
da base
inc
check tens of min.
cmp base+0
lda base+3
inc
tens of min. = 0
cmp base+2
lda base+0
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
cmp base+1
lda base
inc
tens of min.
cmp base+2
lda base
inc
tens of min.
**Initialize interrupt vectors**

```plaintext
org $ff8

tmrvec fdb tmrint
intvec fdb c4int
swiveq fdb start
reset fdb start
```
MC68HC11 EEPROM Programming from a Personal Computer

This application note describes a simple and reliable method of programming either the MC68HC11’s internal EEPROM, or EEPROM connected to the MCU’s external bus. The data to be programmed is downloaded from any standard personal computer (PC) fitted with a serial communications port. In addition to the programming procedure, the software incorporates the facility to verify the contents of the MCU’s internal or external memory against code held on a PC disc. Both program and verify options use data supplied in S record format, which is downloaded from the PC to the MC68HC11 using the RS232 protocol supported by the MCU’s SCI port.

The minimum MCU configuration required to program the MC68HC11’s internal EEPROM is shown in Figure 1. This consists only of the MCU, an RS232 level shifting circuit, plus an 8 MHz crystal and a few passive components.

To initiate the download, the PC is connected to the MC68HC11 SCI transmit and receive lines via a level shifter. The circuit of Figure 1 uses a Maxim MAX232 to eliminate the need for additional ±12 V supplies. The MCU’s special bootstrap mode is invoked by applying a logic zero to the MODA and MODB pins, followed by a hardware RESET.

Removing the RESET condition causes the MCU to start execution of its bootloader program, located in internal ROM, between addresses $8F40 and $FFFF. In normal single-chip or expanded modes, the boot ROM is not accessible, and reads from these memory locations will result respectively in irrelevant data or external memory fetches.

An additional consequence of bootstrap operation is that all vectors are relocated to the boot ROM area. With the exception of the RESET vector, which points to the start of the boot ROM, the remaining interrupt vectors all point to an uninitialized jump table in RAM. Three bytes are reserved for each entry in the jump table, to allow for an extended jump instruction. Tables 1 and 2 detail the memory map of the bootstrap vectors and an example RAM jump table.

**Figure 1. MC68HC11 Bootstrap Mode Connection to RS232 Line**

C1, C2, C3, C4 — 22 μF 25 V Aluminium or Tantalum

NOTE: To improve reliability of the MCU, all its unused inputs should be connected to VSS or VDD.
Illegal Opcode

strap mode, it is the software designer's responsibility to
transmitter and receiver to 7812 baud and proceeds to
MC68HC11's interrupt system, the jump table is not set
examine the state of the
up.
initialize all
loader will erase the entire contents of internal
information of a proprietary or confidential nature.
SEC
Note that, before any interrupts are enabled in boot­
program continues by initializing the
register.
CONFIG
Note also that erasing the CONFIG register disables the
security feature.
The bootstrap program then issues a break condition
on the SCI transmit line, and waits for the reception of
the first byte. In this application, no use is made of the
break transmitted by the SCI.
At this point, it is necessary to initiate the PC S record
downloader program, called ELOAD.BAS (written in
BASIC). It will display a header message, and prompt the
user for the number of the COM channel (either one or
two) which is connected to the MC68HC11. A listing of
ELOAD.BAS is given at the back of this application note.
The PC-resident program will now configure the ap­
propriate COM channel to 1200 baud, one stop bit, no
parity, and download the binary file EEPROGIX.BOO from
the PC to the MC68HC11.
The MC68HC11's bootloader will automatically detect
the fact that the first incoming character is received at a
different baud rate, and change its SCI rate to 1200 baud.
It will then proceed to load the binary file into all 256
RAM locations and then jump to address $0000 (i.e., the
first RAM location).
EEPROGIX.BOO consists of the MC68HC11 executable
code shown in the source listing at the back of this ap­
lication note, with the addition of $FF at the head of the
file, and $00 appended up to the 256th byte. This program
is designed to receive S records from the PC and program
the data fields into the appropriate EEPROM memory
locations.
A point to note is that the initial $FF byte in EEPRO­
GIX.BOO is only used to detect the baud rate of the PC,
and is not echoed back, while the remaining 256 bytes
are echoed back by the MC68HC11's SCI transmitter. However,
during download of EEPROGIX.BOO, the PC does not
detect the echo, as this feature is unnecessary at this
stage.
Once the newly downloaded S record programmer
starts execution in the MC68HC11, it configures the SCI
to 9600 baud, then waits for a control character from the
PC. This character will determine the operating mode
of the S record programmer. The options available are shown
in Table 3. Note that these programming utilities can be
used to load and verify external RAM as well as external
EEPROM.

<table>
<thead>
<tr>
<th>Address</th>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFFE</td>
<td>BF40</td>
<td>Bootstrap Reset</td>
</tr>
<tr>
<td>BBFC</td>
<td>00FD</td>
<td>Clock Monitor</td>
</tr>
<tr>
<td>BFFA</td>
<td>00FA</td>
<td>COP Fail</td>
</tr>
<tr>
<td>BFFB</td>
<td>00F7</td>
<td>Illegal Opcode</td>
</tr>
<tr>
<td>BFF6</td>
<td>00F4</td>
<td>SWI</td>
</tr>
<tr>
<td>BFF4</td>
<td>00F1</td>
<td>XIRQ</td>
</tr>
<tr>
<td>BFF2</td>
<td>00EE</td>
<td>IRO</td>
</tr>
<tr>
<td>BFF0</td>
<td>00EB</td>
<td>Real Time Interrupt</td>
</tr>
<tr>
<td>BFEE</td>
<td>00E8</td>
<td>Timer Input Capture 1</td>
</tr>
<tr>
<td>BFEC</td>
<td>00E5</td>
<td>Timer Input Capture 2</td>
</tr>
<tr>
<td>BFEA</td>
<td>00E2</td>
<td>Timer Input Capture 3</td>
</tr>
<tr>
<td>BFE8</td>
<td>00DF</td>
<td>Timer Output Compare 1</td>
</tr>
<tr>
<td>BFE6</td>
<td>00DC</td>
<td>Timer Output Compare 2</td>
</tr>
<tr>
<td>BFE4</td>
<td>00D9</td>
<td>Timer Output Compare 3</td>
</tr>
<tr>
<td>BFE2</td>
<td>00D6</td>
<td>Timer Output Compare 4</td>
</tr>
<tr>
<td>BFE0</td>
<td>00D3</td>
<td>Timer Output Compare 5</td>
</tr>
<tr>
<td>BFDE</td>
<td>00D0</td>
<td>Timer Overflow</td>
</tr>
<tr>
<td>BFDC</td>
<td>00CD</td>
<td>Pulse Accumulator Overflow</td>
</tr>
<tr>
<td>BFDA</td>
<td>00CA</td>
<td>Pulse Accumulator Input Edge</td>
</tr>
<tr>
<td>BFD8</td>
<td>00C7</td>
<td>SPI</td>
</tr>
<tr>
<td>BFD6</td>
<td>00C4</td>
<td>SCI</td>
</tr>
</tbody>
</table>

Table 1. Bootstrap Vector Assignments

<table>
<thead>
<tr>
<th>Address</th>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFFE</td>
<td>BF40</td>
<td>Bootstrap Reset</td>
</tr>
<tr>
<td>BBFC</td>
<td>00FD</td>
<td>Clock Monitor</td>
</tr>
<tr>
<td>BFFA</td>
<td>00FA</td>
<td>COP Fail</td>
</tr>
<tr>
<td>BFFB</td>
<td>00F7</td>
<td>Illegal Opcode</td>
</tr>
<tr>
<td>BFF6</td>
<td>00F4</td>
<td>SWI</td>
</tr>
<tr>
<td>BFF4</td>
<td>00F1</td>
<td>XIRQ</td>
</tr>
<tr>
<td>BFF2</td>
<td>00EE</td>
<td>IRO</td>
</tr>
<tr>
<td>BFF0</td>
<td>00EB</td>
<td>Real Time Interrupt</td>
</tr>
<tr>
<td>BFEE</td>
<td>00E8</td>
<td>Timer Input Capture 1</td>
</tr>
<tr>
<td>BFEC</td>
<td>00E5</td>
<td>Timer Input Capture 2</td>
</tr>
<tr>
<td>BFEA</td>
<td>00E2</td>
<td>Timer Input Capture 3</td>
</tr>
<tr>
<td>BFE8</td>
<td>00DF</td>
<td>Timer Output Compare 1</td>
</tr>
<tr>
<td>BFE6</td>
<td>00DC</td>
<td>Timer Output Compare 2</td>
</tr>
<tr>
<td>BFE4</td>
<td>00D9</td>
<td>Timer Output Compare 3</td>
</tr>
<tr>
<td>BFE2</td>
<td>00D6</td>
<td>Timer Output Compare 4</td>
</tr>
<tr>
<td>BFE0</td>
<td>00D3</td>
<td>Timer Output Compare 5</td>
</tr>
<tr>
<td>BFDE</td>
<td>00D0</td>
<td>Timer Overflow</td>
</tr>
<tr>
<td>BFDC</td>
<td>00CD</td>
<td>Pulse Accumulator Overflow</td>
</tr>
<tr>
<td>BFDA</td>
<td>00CA</td>
<td>Pulse Accumulator Input Edge</td>
</tr>
<tr>
<td>BFD8</td>
<td>00C7</td>
<td>SPI</td>
</tr>
<tr>
<td>BFD6</td>
<td>00C4</td>
<td>SCI</td>
</tr>
</tbody>
</table>

Note that, before any interrupts are enabled in boot­
strap mode, it is the software designer's responsibility to
initialize all appropriate entries in the jump table.
As this application note does not make use of the
MC68HC11’s interrupt system, the jump table is not set
up.
The bootstrap program continues by initializing the SCI
transmitter and receiver to 7812 baud and proceeds to
examine the state of the NOSEC bit in the CONFIG regis­
ter. If this is at logic zero (security enabled) the boot­
loader will erase the entire contents of internal EEPROM
and also the CONFIG register.
This feature is particularly useful for security conscious
applications, where the internal EEPROM contains informa­
tion of a proprietary or confidential nature. If the NO­
SEC bit is at logic one, then the erasing sequence is not
carried out.

Table 2. RAM Jump Table

<table>
<thead>
<tr>
<th>Address</th>
<th>Typical Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>00FD</td>
<td>JMP CLMKMON</td>
</tr>
<tr>
<td>00FA</td>
<td>JMP COPFL</td>
</tr>
<tr>
<td>...etc</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. S Record Downloader Operating Mode Options

<table>
<thead>
<tr>
<th>Control Character</th>
<th>Operating Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Program External EEPROM RAM</td>
</tr>
<tr>
<td>I</td>
<td>Program Internal EEPROM</td>
</tr>
<tr>
<td>V</td>
<td>Verify Internal or External EEPROM</td>
</tr>
</tbody>
</table>

If the S record programmer has been downloaded suc­
cessfully, the PC resident program will now —
1. Request whether the downloaded data must be
echoed to the screen.
2. Prompt the user for the required operating mode.
3. Request the name of the S record file to be down­
loaded from the PC.

386
Once the download starts, every character in the S record file is immediately echoed back to the PC. This ensures synchronism between the PC and the MC68HC11, and at the same time, removes some of the overhead associated with the EEPROM programming delay time. It also removes the need for a hardware handshake.

**VERIFY OPTION**

If a verify error occurs, the actual stored byte value is returned to the PC, where it is displayed with a preceding colon delimiter. In this way, EEPROM data and address faults can be quickly identified by inspection. At the end of the verify download, the total number of errors is displayed.

**INTERNAL OR EXTERNAL OPTION**

If a programming error occurs in either internal or external programming mode, i.e., if the read back data after programming does not correspond to the expected data, the MC68HC11-resident software will hang up. This condition is detected by the PC-resident program, which will then abort the download and display an error message. This same error message is displayed if a fault or incorrect connection exists on the serial link between the PC and MC68HC11.

There is one exception to this operation. It stems from the fact that changes to the MC68HC11's CONFIG register can only be detected after a subsequent hardware RESET. If the CONFIG register address ($103F) is detected, then the CONFIG register is not read directly after programming. This prevents premature termination of the download.

To allow programming of the CONFIG register in all mask set versions of the MC68HC11A series, and to permit expanded mode operation, the MCU resident program switches from bootstrap mode to special test mode, by setting the MDA bit (bit 5) in the HPRIO register (address $103C).

If the user wishes to maintain operation in bootstrap mode, (to verify internal ROM code, for instance), then the 'BSET HPRIO,X,#MDA' instruction on the 8th line of program code in EEPROGIX.ASC should be removed, and the program reassembled.

**PROGRAMMING INTERNAL EEPROM**

The techniques for programming internal and external EEPROM are quite different. With internal EEPROM, it is first generally necessary to erase the required byte (erased state is $FF), and follow with a write of data to the same address.

The internal programming sequence involves accessing the PPROG register (address $103B) to latch the EEPROM address and data buses for the duration that the programming voltage is applied. Also, the programming time delay must be implemented or initiated by software. In this application, a software timing loop is used, but one of the internal MC68HC11 timer functions could equally well be used to provide the time delay.

Figures 2 and 3 show the flowcharts of the internal EEPROM erase and write sequences.

---

**Figure 2. Internal EEPROM Erase Sequence**
Figure 3. Internal EEPROM Write Sequence

PROGRAMMING EXTERNAL EEPROM

Figure 4 shows the hardware needed to interface the MC68HC11 to an external 2864 EEPROM, which provides a total of 8K bytes of reprogrammable memory. The addition of the MC68HC24 gives a minimal component count implementation of a circuit which accurately emulates the MC68HC11A8 single-chip MCU. The added benefit of using the 2864 is that the software designer’s program and/or data can be modified without removing the emulator from the target system. This can be particularly useful in applications where the emulator may be enclosed in a confined space or in an environmental chamber.

To program the 2864 from the PC, the external operating mode option (X) must be selected from the EELOAD menu.

Programming the 2864 involves fewer operations than are needed for internal EEPROM, as the former has no equivalent of the PPROG control register. In addition, the erase sequence and delay time are handled automatically by the 2864 on-chip logic.

A data polling technique is used to determine the end of the programming delay time. This involves examining the most significant bit of the data programmed, by reading from the address just written to, until the data becomes true. (During the programming delay time, the M5 bit will read as the complement of the expected data).

This means that the same software algorithm can be used to download code or data to external RAM as well as external EEPROM.

EMULATOR ADDRESS DECODING

The emulator circuit in Figure 4 shows the MC68HC11’s address line A13 connected to pin 26 of the 2864. Though this pin is actually unused by the 2864, its inclusion permits the replacement of the 2864 with a 27128 16K byte EEPROM memory.

An important outcome of this is that, when a 2864 is used, the memory range $C000-$DFFF is mapped over the normally used 8K byte range of $E000-$FFFF. In practice, this should never pose a problem. When a 27128 memory is used, its full 16K byte address range of $C000-$FFFF is available to the MCU.

Included in the S record programmer, irrespective of the selected programming mode, is a feature to force program execution at the address specified in the S9, S record address field, provided the address is not $0000.

Figure 5 shows the general format of S record files.
Figure 4. MC68HC11A8 Emulator Using 2864 EEPROM
RECORD TYPE: S0, S1, OR S9

S0 — HEADER RECORD: LOAD ADDRESS FIELD $0000. CODE/DATA
FIELD CONTAINS OPTIONAL DESCRIPTIVE INFORMATION.
S1 — CODE/DATA RECORD. CODE/DATA FIELD CONTAINS EXECUTABLE CODE OR DATA.
S9 — TERMINATION RECORD: LOAD ADDRESS FIELD CONTAINS OPTIONAL EXECUTION ADDRESS. THERE IS NO
CODE/DATA FIELD, JUST A CHECKSUM.

NUMBER OF HEX CHARACTERS FOLLOWING (Len) $14
MEMORY LOAD ADDRESS FIELD (Ldhi, Ldllo) $C01E

S1 14 301E0829BC02A18396A3B6F3B91B09826FC39 OF

CODE/DATA FIELD
CHECKSUM BYTE
0 TO N BYTES OF CODE/DATA

APART FROM THE LETTER S AT THE START, ALL CHARACTERS IN THE RECORDS ARE HEXADECIMAL DIGITS REPRESENTED IN ASCII FORMAT.

CHECKSUM ALGORITHM: LSB OF

\[ \text{LSB of } \left( \text{Len} \times \text{Ldhi} \times \text{Ldllo} \times \sum_{k=0}^{n} \text{byte}_k \right) \]

NOTE: The S-record programmer in this application ignores the checksum byte.

Figure 5. S-Record Format
This program downloads S record file to the MC68HC11 through special bootstrap program, designed to program either internal or external EEPROM in the 68HC11's memory map. The loader can also verify memory against an S record file. Downloaded data is optionally echoed on terminal.

First make sure loader program is available.

ON ERROR GOTO 880

OPEN LOADERS FOR INPUT AS 112

CLOSE 112

ON ERROR GOTO 0

CHANS="0"

WHILE CHANS<"1" AND CHANS<"2"

GOSUB 1070

LINE INPUT "Enter COM channel number (1/2):",CHANS

WEND

CMS="COM"+CHANS

' Now set baud rate to 1200 and load EEPROG through boot loader'

' by executing DOS MODE and COPY commands'

SHELL "MODE "+CMS"=":1200,N,8,1"

SHELL "COPY "+LOADERS+" "+CMS

GOSUB 1070

FOR I=1 TO 4:PRINT CLRLNS$:NEXT I:PRINT: 'Clear DOS commands from screen'

ECHO=''

WHILE ECHOS<"Y" AND ECHOS<"N"

GOSUB 1070

LINE INPUT "Do you want echo to screen (Y/N):",ECHOS

WEND

ROW=CSRLIN: 'Store current line number'

WHILE EEOPTS<"X" AND EEOPTS<"I" AND EEOPTS<"V"

GOSUB 1070

LINE INPUT "Select Internal,External or Verify EEPROM option (I/X/V):",EEOPTS

WEND

OPTS="Verify"

IF EEOPTS="I" THEN OPTS="Internal"

IF EEOPTS="X" THEN OPTS="External"

ROW=CSRLIN: 'Store current line position in case of file error'

RXERR=0: 'Initialise number of RX errors allowed'

ON ERROR GOTO 910

GOSUB 1070
600 IF OPTS="Verify" THEN INPUT "Enter filename to verify: ",FS ELSE INPUT "Enter filename to download:",FS
610 CLOSE
620 OPEN FS FOR INPUT AS #2
630 ON ERROR GOTO 0
640 'COM1 or 2 connected to SCI on HC11'
650 OPEN CMS+";9600,N,8,1" AS #1
660 'Establish contact with HC11 by sending CR char & waiting for echo'
670 ON ERROR GOTO 660: 'Clear potential RX error'
680 PRINT #1,CRS;
690 GOSUB 990: 'Read char into BS'
700 'Transmit Internal,External or Verify EEPROM option char to 68HC11'
710 PRINT #1,EEOPTS;GOSUB 990: 'No echo to screen'
720 ON ERROR GOTO 930
730 PRINT "Starting download of ";FS;" to: ";OPTSi" Eeprom"
732 IF ECHOS="Y" THEN EX=1 ELSE EX=0
734 IF EEOPTS="Y" THEN VX=1 ELSE VX=0
740 'WHILE NOT EOF(2)
750 INPUT #2,SS
751 LX=LEN(SS)
752 FOR IX=1 TO LX
760 PRINT I,MIDS(SS,IX,1);:GOSUB 990:lf EX
770 If VX
780 PRINT "Download Complete"
785 NEXT IX
787 IF ECHOS="Y" THEN PRINT 
790 IF VX THEN PRINT 
795 PRINT 800 PRINT "Download Complete"
810 IF VX THEN PRINT ERRTOTX;" error(s) found"
820 CLOSE #2
830 SYSTEM
840 END
850 ' ---------------
860 IF RXERR>5 THEN 940 ELSE RXERR=RXERR+1:RESUME 610
870 ' ---------------
880 PRINT:PRINT ERMS;LOADERS:PRINT "Program aborted"
890 GOTO 830
900 ' ---------------
910 PRINT ERMS;FS;SPACES(40)
920 RESUME 580
930 ' ---------------
940 PRINT:PRINT "Communication breakdown: Download aborted"
950 GOTO 820
960 ' ---------------
970 '---SUB waits for received character, with time limit
980 ' returns with char in BS, or aborts if time limit exceeded
990 TOX=O:WHILE LOC(1)=O:IF TOX>100 THEN 940 ELSE TOX=TOX+1:WEND
1000 BS=INPUTS(1,1):RETURN
1010 ' ---------------
1020 '---SUB waits for received character, with time limit
1025 ' returns with char in CS, or null in CS if time limit exceeded
1030 TOX=O:CS="":WHILE LOC(1)=O AND TOX<1:TOX=TOX+1:WEND
1040 IF LOC(1)>0 THEN CS=INPUTS(1,1):ERRTOTX=ERRTOTX+1
1050 RETURN
1060 ' ---------------
1070 '---SUB Clear line 
1080 LOCATE ROW,1,1:PRINT CLRNS
1090 LOCATE ROW,1,1:RETURN
1100 ' ---------------
This program loads S records from the host to either a 2864 external EEPROM on the 68HC11 external bus, or to the 68HC11's internal EEPROM. It can also be used to verify memory contents against an S record file or just load RAM located on the 68HC11's external bus.

Each byte loaded is echoed back to the host.

When programming a 2864, data polling is used to detect completion of the programming cycle.

As the host software always waits for the echo before downloading the next byte, host transmission is suspended during the data polling period.

Because the serial communication rate (~1ms/byte) is slower than the 2864 internal timer timeout rate (~300us), page write mode cannot be used. This means that data polling is active on each byte written to the EEPROM, after an initial delay of approx 500us.

When the internal EEPROM is programmed, instead of data polling, each byte is verified after programming. In this case, the 500us delay is not required and is bypassed.

If a failure occurs, the program effectively hangs up. It is the responsibility of the host downloader program to detect this condition and take remedial action.

The BASIC program EELOAD just displays a 'Communication breakdown' message, and terminates the program.

When used in the verify mode, apart from the normal echo back of each character, all differences between memory and S record data are also sent back to the host.

The host software must be capable of detecting this, and perform the action required.

The BASIC loader program EELOAD simply displays the returned erroneous byte adjacent to the expected byte, separated by a colon.

Before receiving the S records, a code byte is received from the host. i.e.: ASCII 'X' for external EEPROM, ASCII 'I' for internal EEPROM, ASCII 'V' for verify EEPROM.

This program is designed to be used with the BASIC ELOAD program.

Data transfer is through the SCI, configured for 8 data bits, 9600 baud.
55 A * Constants
56 A 0080 TDRE EQU $80
57 A 0020 RDIF EQU $20
58 A 0020 NDA EQU $20
59 A 0040 SMOD EQU $40
60 A 0005 mS10 EQU 10000/3 10mS delay with 8MHz xtal.
61 A 00A6 US500 EQU 500/3 500uS delay.
62 A *
63 A * Registers
64 A 0028 BAUD EQU $28
65 A 002C SCCR1 EQU $2C
66 A 0020 SCCR2 EQU $2D
67 A 002E SCSR EQU $2E
68 A 002F SCDR EQU $2F
69 A 0038 PPRG EQU $38
70 A 003C HPRIO EQU $3C
71 A 013F CONFIG EQU $103F
72 A *
73 A * Variables. Note: They overwrite initialisation code!!!!
74 A 0000 ORG $0
75 P 0000 0001 EE0PT RMB 1
76 P 0001 0001 MASK RMB 1
77 P 0002 0001 TEMP RMB 1
78 P 0003 0001 LASTBYTE RMB 1
79 A *
80 A *
81 A 0000 ORG $0
82 A 0000 000F BSET EEOPT, #1000 Offset for control registers.
83 A 0003 CE1000 LDX #1000 Initialise SCI for 8 data bits, 9600 baud
84 A 0006 6F2C CLR SCCR1,X Initialise SCI for 8 data bits, 9600 baud
85 A 0008 CC300C LDD #$300C
86 A 0008 A72B STAA BAUD,X
87 A 0000 672D STAB SCCR2,X
88 A 0000 1C3C20 BSET HPRIO,X,#MDA Force Special Test mode first,
89 A *>>> MAINTAIN SPECIAL TEST MODE TO ALLOW B960 CONFIG REGISTER PROGRAMMING <<<
90 A * BCLR HPRIO,X,#SMOD and then expanded mode. (From Bootstrap mode)
91 A 0012 9F00 ReadOpt STS <EE0PT Default to internal EEPROM: EE0PT=0; MASK=$FF;
92 A 0014 BD7C BSR READC Then check control byte for external or internal
93 A 0016 C149 CMPB #'1' EEPROM selection.
94 A 0018 2714 BEQ LOAD
95 A 001A C158 CMPB #'X' If external EEPROM requested
96 A 001C 2609 BNE OptVerf
97 A 001E 7C0000 INC EE0PT then change option to 1
98 A 0021 8680 LDAA #$80
99 A 0023 9701 STAA <MASK and select mask for data polling mode.
100 A 0025 2007 BRA LOAD
101 A *
102 A 0027 C156 OptVerf CMPB #'V' If not verify then
103 A 0029 26E7 BNE ReadOpt get next character else
104 A 002B 7A0000 DEC EE0PT make EE0PT flag negative.
105 A *
106 A 002E LOAD EQU *
107 A 002E BD62 BSR READC
108 A 0030 C153 CMPB #'S'
109 A 0032 26FA BNE LOAD
110 A 0034 BD5C BSR READC
111 A 0036 C131 CMPB #'1'
112 A 0038 2719 BEQ LOAD1

394
Complete reading 59 record before terminating

# of bytes to read including checksum.

Get execution address in Y

Now discard remaining bytes, including checksum.

If execution address = 0 then

hang up else

jump to it!

Read byte count of S1 record into ACCB

and store in ACCA

Remove load address & checksum bytes from count

Get load address into X register.

Adjust it for first time thru LOAD2 loop.

Update CC register

If not verifying EEPROM then

If programming external EEPROM

then wait 500uS max.

Now either wait for completion of programming cycle by testing MS bit of last data written to memory or just verify internal programmed data.

When all bytes done,

get next 5 record (discarding checksum) else read next data byte into ACCB.

Advance to next load address

If verifying, then don't program byte!

If internal EEPROM option selected then program else just store byte at address.

Save it for DATA POLLING operation.

If programmed byte is correct then

read next byte

else send bad byte back to host

before reading next byte.

ACCA, X, Y regs unchanged by this routine.

ACCA, X, Y regs unchanged by this routine.

Read next char

and echo it back to host.

Return with char in ACCA.

1st read MS nibble

Convert to binary

and move to upper nibble
Get ASCII char in ACCB

Return with byte in ACCB

Save byte counter

Read MS byte of address and put it in MS byte of ACCD

Now read LS byte of address into LS byte of ACCD

Put load address in Y

Restore byte counter and return.

If ACCB>9 then assume its A-F

Save ACCA.

Default to byte erase mode

If byte's address is CONFIG then use bulk erase, to allow for A1 & A8 as well as A2.

Now erase byte, or entire memory + CONFIG.

Now program byte.

If byte was CONFIG register then load ACCB with old value, to prevent hangup later.

Restore ACCA and return to main bit.

Enable internal addr/data latches.

Write to required address

Enable internal programming voltage

Disable internal programming voltage

Release internal addr/data latches and return

END
<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>ENTRY</th>
<th>Meaning</th>
<th>ENTRY</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUD</td>
<td>002B</td>
<td>PROGA</td>
<td>00CE</td>
<td></td>
</tr>
<tr>
<td>CONFIG</td>
<td>103F</td>
<td>PROGRAM</td>
<td>00E0</td>
<td></td>
</tr>
<tr>
<td>DATAPOLL</td>
<td>0069</td>
<td>PROGX</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>EEPROM</td>
<td>0000</td>
<td>RDBYTE</td>
<td>009F</td>
<td></td>
</tr>
<tr>
<td>GETADR</td>
<td>0080</td>
<td>RDRF</td>
<td>0020</td>
<td></td>
</tr>
<tr>
<td>HEXBIN</td>
<td>008A</td>
<td>READC</td>
<td>0092</td>
<td></td>
</tr>
<tr>
<td>HEXNUM</td>
<td>00C0</td>
<td>ReadOpt</td>
<td>0012</td>
<td></td>
</tr>
<tr>
<td>HPRI0</td>
<td>003C</td>
<td>SCCR1</td>
<td>002C</td>
<td></td>
</tr>
<tr>
<td>LASTBYTE</td>
<td>0003</td>
<td>SCCR2</td>
<td>0020</td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>002E</td>
<td>SCSR</td>
<td>002F</td>
<td></td>
</tr>
<tr>
<td>LOAD1</td>
<td>0053</td>
<td>SCSR</td>
<td>002E</td>
<td></td>
</tr>
<tr>
<td>LOAD1A</td>
<td>005E</td>
<td>SMOD</td>
<td>0040</td>
<td></td>
</tr>
<tr>
<td>LOAD1B</td>
<td>0075</td>
<td>TDRE</td>
<td>0080</td>
<td></td>
</tr>
<tr>
<td>LOAD1D</td>
<td>0083</td>
<td>TEMP</td>
<td>0002</td>
<td></td>
</tr>
<tr>
<td>LOAD1E</td>
<td>0072</td>
<td>VERIFY</td>
<td>0087</td>
<td></td>
</tr>
<tr>
<td>LOAD9</td>
<td>0045</td>
<td>WAIT1</td>
<td>0066</td>
<td></td>
</tr>
<tr>
<td>MASK</td>
<td>0001</td>
<td>WAIT2</td>
<td>00EB</td>
<td></td>
</tr>
<tr>
<td>MDA</td>
<td>0020</td>
<td>WRITEC</td>
<td>0098</td>
<td></td>
</tr>
<tr>
<td>OptVerf</td>
<td>0027</td>
<td>mS10</td>
<td>0005</td>
<td></td>
</tr>
<tr>
<td>PPROG</td>
<td>0038</td>
<td>US500</td>
<td>00A6</td>
<td></td>
</tr>
<tr>
<td>PROG</td>
<td>00C3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DESIGNING FOR ELECTROMAGNETIC COMPATIBILITY (EMC) WITH HCMOS MICROCONTROLLERS

by
Mike Catherwood
Motorola Inc.
Austin, Tx.

INTRODUCTION

The operating speed of present HCMOS devices is approaching that of the fastest bipolar logic families of only a few years ago. Associated with this increase in performance are some new design challenges for the MCU-based system designer. This application note addresses one of these issues, the electromagnetic compatibility (EMC) of the finished product. EMC may be considered from either an emission or a susceptibility point of view. Although the following discussion relates primarily to emission control (in particular, radiated emission), most techniques to limit emission also reduce susceptibility. Furthermore, minimizing electromagnetic interference (EMI) will reduce overall system noise, the benefits of which are higher digital noise immunity and accurate operation of local analog subsystems — i.e., better design margin and a more reliable end product.

EMC can only exist when the system functions correctly within the intended electromagnetic environment and does not exceed the EMI levels specified in the appropriate standards documents. EMI, which encompasses interference in a bandwidth of ‘dc to daylight’ is a generalization of a much older term, radio frequency interference (RFI), which is now defined to encompass 10 kHz to 3 GHz. Failure to consider EMC during early phases of the design process may result in expensive modifications (possibly with many additional components), printed circuit board (PCB) re-layout, product introduction delays, and EMC consultant fees to conform to the required standards.
LEGAL REQUIREMENTS

The Federal Communications Commission (FCC) have a set of standards to regulate EMI in electronic equipment and systems for use in the United States. Compliance with the appropriate sections of these regulations is mandatory to market or sell a product except for certain subclasses of digital devices that are temporarily exempt. Engineering models (including field-trial prototypes which are not sold) are also exempt; however, the display of a product at an electronics show is considered a marketing function subject to regulations.

FCC rules and regulations (part 15, subpart J of Title 47 of The Federal Regulations) apply to almost all digital devices (see Reference 1), defining standards and operational requirements for all devices capable of emitting RF energy within the range 450 kHz to 1 GHz.

Equipment for use within West Germany must comply with a different set of standards defined and administered by the Verland Deutcher Electro-Techniker (VDE). Digital equipment is generally required to meet both VDE0871 standards. In other countries, compliance to a standard is not always mandatory, however, the European Economic Community (EEC) member states intend to introduce a mandatory RFI performance standard after 1st January 1992. The current proposal is based on International Special Committee on Radio Interference specification CISPR22 and is referred to as European norm EN55022. As the FCC is a member of the CISPR, and has voted in favor of the CISPR22 standard, it is likely that the FCC will ultimately adopt the same standards. CISPR22 is somewhat more stringent than FCC part 15, subpart J in the 88 to 230 MHz frequency range, though it is less stringent than some aspects of the VDE0871.

RFI PROBLEM OVERVIEW

The frequency spectrum of a periodic waveform has been shown, through Fourier analysis, to be composed of discrete frequency components that include the fundamental (fo) and multiple harmonics (n × fo). For a typical trapezoidal waveform, the relative amplitude of each frequency component is related to the fundamental frequency, the rise time, and mark-to-space ratio (duty cycle) of the waveform (see Reference 2). Doubling the frequency, halving the rise time, or halving the mark-to-space ratio will double (+6 dB) the amplitude of a specific harmonic frequency.
It is possible to graphically predict the harmonic spectrum of a specific trapezoidal waveform by plotting the amplitude of two corner frequencies and a reference (0 dB) point. This plot is referred to as a Fourier envelope, a Bode plot, or a nomogram. An example is shown in Figure 1 where:

\[
\begin{align*}
0 \text{ dB reference} &= 20 \log(2A\delta) \quad \text{dB} \\
f_1 &= \frac{1}{P} \quad \text{Hz} \\
f_2 &= \frac{1}{T_{\text{r}}} \quad \text{Hz}
\end{align*}
\]

where:

- \( V \) = amplitude \( \text{V} \)
- \( P \) = pulse width \( \text{s} \)
- \( t_{\text{r}} \) = rise time \( \text{s} \)
- \( T \) = period \( \text{s} \)
- \( \delta = (P - t_{\text{r}})T \)

At frequencies beyond \( f_1 \) (\( 1/P \)), the amplitude of the harmonics falls off at \(-20 \text{ dB/decade}\). Above \( f_2 \) (\( 1/T_{\text{r}} \)), the amplitude of the harmonics falls off at \(-40 \text{ dB/decade}\). For many applications, these latter harmonics are considered small enough to be ignored; thus, the bandwidth of a system is generally defined to be \( 1/T_{\text{r}} \). For example, an HCMOS device which can produce an external periodic signal with edge times on the order of 2 ns can generate significant harmonics (i.e., have a bandwidth) of up to 160 MHz. Any PCB tracks, component leads, cables, or connectors attached directly or capacitively to signal sources, such as those previously described, can act as antennas and radiate the harmonics with varying degrees of efficiency. Radiated emission from a system may be either differential-mode or common-mode radiation; common-mode radiation is typically more difficult to reduce.

**DIFFERENTIAL-MODE RADIATION**

Differential-mode radiation is caused by the flow of RF current loops around the system conductors. For a small loop area, the far-field electric term, when operating in an field above a ground plane (free space is not a typical environment), can be shown to be approximately (see Reference 3):

\[
E = 2.6(A I_L f^2)/R \quad \mu\text{V/m}
\]

where:

- \( A \) = loop area \( \text{cm}^2 \)
- \( I_L \) = loop current \( \text{A} \)
- \( f \) = frequency \( \text{MHz} \)
- \( R \) = distance \( \text{m} \)
Figure 1. Nomogram of a Trapezoidal Waveform
For a constant current and loop area, the electric field at a prescribed distance is proportional to the square of the frequency (i.e., it increases at 40 dB/decade). Adding this term to the Fourier envelope indicates that the differential-mode radiated emission increases at 20 dB/decade up to $f_2$, after which it remains flat. $R$ is fixed by both the FCC and VDE rules and regulations, and $f$ is usually not a system variable; however, $A$ and $I_L$ can be reduced through thoughtful board layout and careful circuit design.

COMMON-MODE RADIATION

Common-mode (CM) radiation is caused by unintentional voltage drops in a circuit, which cause some grounded parts of the circuit to rise above the real ground potential (see Figure 2). Cables connected to the affected ground act like antennas and radiate the components of the CM potential. The far-field electric term can be shown as follows (see Reference 3):

$$E = \frac{(f \cdot I_{CM} \cdot L)}{R} \text{ V/m}$$

(2)

where:

- $L =$ antenna length \( \text{m} \)
- $I_{CM} =$ common-mode current \( \text{A} \)
- $f =$ frequency \( \text{Hz} \)
- $R =$ distance \( \text{m} \)

For a constant current and antenna length, the electric field at a prescribed distance is proportional to the frequency (i.e., it increases at 20 dB/decade). Adding this term to the Fourier envelope indicates that the CM radiated emission remains flat up to $f_2$, then decreases at $-20 \text{ dB/decade}$ for frequencies above $f_2$. Unlike differential-mode radiation, which is relatively easy to reduce through careful product design, CM radiation is more difficult to control since the only variables available to the designer are typically the common path impedances and CM current. Obviously, to eliminate the radiation, the CM current must approach zero, which can be achieved through a sensible grounding scheme and the addition of inductors or capacitors to increase the cable (antenna) impedance.

SUPPLY DECOUPLING

Inadequate decoupling decreases system noise margins and ultimately leads to incorrect, unreliable, or unstable operation. For example, the MC68HC11A8 can generate peak supply-current transients of approximately $100 \text{ mA}$, which is typical of an HCMOS microcontroller. Although the average supply current is only a few milliamps, the power supply must be able to source the peak supply-current levels
(a) Poor Supply Decoupling

(b) Equivalent Circuit of (a)

Figure 2. PCB Layout
to guarantee correct operation. Also, for fast digital logic, the peak supply-current
transients are large enough to create an EMI problem if the decoupling layout is
poor.

A decoupling network is used to reduce the supply impedance at the device. To
calculate the value of a decoupling capacitor, the acceptable supply ripple must
first be determined. An appropriate goal is to achieve a maximum ripple of 20% of
the minimum noise immunity voltage — e.g., for the MC68HC11A8 with \( V_{DD} = 5 \) V
and no loads:

\[
\begin{align*}
    V_{IL} - V_{OL} &= (0.2 \times V_{DD}) - 0.1 = 0.9 \text{ V} \\
    V_{OH} - V_{IH} &= 4.9 - (0.7 \times V_{DD}) = 1.4 \text{ V}
\end{align*}
\]

Therefore,

- Minimum noise immunity = 0.9 V
- Maximum ripple = 0.2 \times 0.9 = 180 \text{ mV}
- Transient period = 10 ns

Now,

\[
C = \frac{I_{DD}}{(dv/dt)} = 100 \text{ mA} (180 \text{ mV/10 ns}) = 0.006 \text{ \mu F}
\]

Rounding up to the nearest preferred value gives 0.01 \( \mu \)F. When operating the
device in expanded mode, the transient currents generated by bus switching can
be significantly larger. Consequently, the recommended decoupling configuration
is a 1 \( \mu \)F tantalum in parallel with a high-frequency 0.01 \( \mu \)F multilayer ceramic (or
similar) capacitor. The parallel 0.01 \( \mu \)F capacitor extends the upper frequency re-
sponse of the network which migh otherwise be reduced due to the internal in-
ductance of the 1 \( \mu \)F capacitor. However, with the exception of VLSI devices,
decoupling capacitors rarely need to exceed 0.01 \( \mu \)F per device. It is also recom-
mended to bulk decouple the board at the supply-line entry point with a 10–100
\( \mu \)F capacitor, depending upon the total board-supply requirements. Because it is
desirable to prevent unwanted supply noise from going off-board and radiating
from the connecting cables, a ferrite bead can be added between the decoupling
capacitor and the connector. Care must be taken to ensure that the DC current will
not saturate the ferrite, making it ineffective.

For a decoupling network to operate successfully, the impedance between the
network and its load must be very low, and, to reduce EMI, its loop area must be
as small as possible. Consider the PCB layout of Figure 2(a); the equivalent circuit
is shown in Figure 2(b).
For DC current,
\[ V_{\text{drop}} = (77) \times 0.1 \text{ mV} \]
\[ = 7.7 \text{ mV} \]

For AC current, assuming 100 mA peak current with a minimum rise time of 10 ns,
Total inductance = 89 nH
\[ V_{\text{drop}} = L \frac{\text{di}}{\text{dt}} \]
\[ = 89 \text{ nH}(100 \text{ mA}/10 \text{ ns}) \]
\[ = 890 \text{ mV} \]

A drop of 0.9 V between the decoupling network and the MCU exceeds the maximum acceptable ripple, even if the recommended network is used. As shown in this example, for fast current transients containing many high-frequency components, the circuit inductance is by far the most critical factor when considering decoupling effectiveness.

Parasitic loop impedance can be effectively reduced through the use of thicker PCB tracks, ground/supply planes, and more direct routing. Decoupling networks should be located as close as possible to the device supply pins. Surface-mount capacitors, which have lower inductance than their leaded counterparts, may be used to the full advantage as decouplers if mounted on the noncomponent side of a PCB across a component, which is the closest possible location. Reducing loop impedance also tends to reduce loop area, which has been previously shown (see Equation (1)) to be directly proportional to radiated field strength.

**SELF-RESONANCE**

The inductance and capacitance within the decoupling loop, essentially results in a series-resonant tuned circuit where:

resonant frequency, \( f = \frac{1}{2\pi} \sqrt{\frac{1}{LC}} \) Hz \hspace{1cm} (3)

At frequencies above \( f \), the impedance of the circuit becomes inductive and results in a less effective decoupler. At resonance, \( f \), the impedance is purely resistive and at a minimum, which can be used to advantage in solving narrow-band RFI problems by tuning suspect decoupling networks to resonate at the problem frequency. For example, to reduce harmonics in the area of 100 MHz (the AM radio band) for a total loop inductance of 10 nH, equate Equation (3) to 100 MHz and solve for \( C \). In this example, \( C \) would equal approximately 250 pF.
LINE TERMINATION

A signal will propagate down a PCB track at approximately 0.6 the speed of light (0.6 ft ns) until it reaches a load. If the line is unterminated (e.g., a high-impedance input), then the degree of impedance mismatch between the load and the line will cause a proportional amount of the signal to be reflected back down the line toward the source. These reflections can induce ringing and overshoot, causing significant EMI problems. If the load equals the characteristic impedance of the line, Z₀, then from viewpoint of the line, the load looks like an infinite line and nothing will be reflected.

In the case of a mismatched line, if the source-signal rise time is sufficiently slow with respect to the line propagation time, then the reflections will be absorbed by the source during the signal rise time. In all other cases, the line should be treated as a transmission line and terminated accordingly (see Reference 3). As a general guide, there should be no need to terminate a line if the one-way propagation delay of a line is less than one-fourth of the signal rise time. For example, for HCMOS with a rise time of 10 ns, the maximum unterminated line length can be estimated as follows:

\[
t_{\text{delay}} < 0.25 \times 10 \text{ ns} \\
< 2.5 \text{ ns}
\]

\[
\text{length} < \text{velocity} \times t_{\text{delay}} \\
< 0.6 \times 2.5 \\
< 1.5 \text{ ft}
\]

Therefore, for the majority of cases, termination will not be necessary when using HCMOS devices. Applying the same criteria to Schottky TTL, which has rise times on the order of 3 ns, provides a maximum length of 5.5 in.

FERRITE BEADS

Ferrite beads have excellent high-frequency characteristics and are especially effective in damping high-frequency switching transients or parasitic ringing due to line reflections. Their low impedance (usually below 100 Ω) makes them particularly suitable to filter out supply noise above approximately 1 MHz, preventing the noise from going off-board or into another circuit. However, care must be taken to ensure that the DC current does not saturate the ferrite if it is to be an effective filter. Ferrites having a variety of characteristics are available in many different packages, including surface mount.
GROUNDBNG TECHNIQUES

A ground is supposed to be an equipotential point or plane used as a reference potential within a system. In reality, this is untrue due to inevitable parasitic inductance and high ground currents causing significant voltage drops, which can result in common-mode radiation problems. To design a successful grounding scheme, the designer must be aware of the paths that ground currents will take to identify possible common-mode impedance problems, reduce loop areas, and prevent noisy return currents from interfering with low-level circuits.

Signal grounds can be classified as single-point, multipoint, or hybrid grounds (see Figure 3). Single-point is acceptable for low frequencies but may have too much impedance at higher frequencies to operate correctly. The ground wire length should be kept as short as possible to reduce inductance and radiating ability. A multipoint ground is used in high-frequency systems, such as digital circuitry, in which each element is connected to the nearest low-impedance ground plane. A hybrid ground looks like a single-point ground at low frequencies and a multipoint ground at high frequencies. A typical system is often a mixture of grounding techniques.

Figure 4 shows a typical MCU application grounding scheme, categorized into low-level analog, digital, input/output (I/O) buffer, high-current switching, and hardware grounds. A single-point ground is located at the source of primary power, which is typically the power supply. The on-board digital logic has a multipoint ground, though it is grounded off-board through a single-point ground. To prevent radiation, no high-frequency components of digital return current should be allowed off-board; thus, the board power-supply lines should only carry DC current, which is suitable for single-point grounding. A block diagram, such as the one shown in Figure 4, is a useful starting point for the design of a good grounding scheme.

ANALOG-DIGITAL MIX

Combining analog and digital circuitry onto a single board requires special attention to PCB layout. Figure 5(a) demonstrates how common-mode impedance ground coupling can superimpose noise on an analog input signal. For example, if the analog section were a 12-bit A/D converter, the added digital noise would significantly reduce the achievable accuracy of the measurements, possibly by several bits. In Figure 5(a), the analog circuit shares its ground and supply with the noisy digital section and is therefore within the digital supply loop. The PCB tracks are also very thin, increasing the parasitic inductance and voltage drop. A better layout of the board is shown in Figure 5(b) in which the digital supply and ground tracks are substantially wider and the analog circuitry is provided with its own supply and ground reference. Any voltage drop occurring on the digital ground track no longer affects the analog input signal because the digital current no longer passes through the analog input loop.
Figure 3. Grounding Techniques
Figure 4. Typical MCU Application Grounding Example
Figure 5. Analog Circuit Grounding
Adequate supply decoupling is also a prerequisite to minimizing noise in an analog subsystem. With regard to the MC68HC11 on-chip A/D converter, the recommended decoupling network for the analog reference inputs is shown in Figure 6.

![Decoupling Network Diagram]

*Vrefl SHOULD NOT GO BELOW VSS–0.6 V

**Figure 6. Recommended A/D Reference Voltage Decoupling for the MC68HC11**

**I/O CABLES AND SHIELDING**

Providing a low-noise ground for I/O enables I/O shunt filters to be used to remove common-mode voltages from I/O cables that extend beyond the enclosure (see Figure 4). In addition, externally shielding I/O cables is ineffective if the termination grounds are themselves noisy. Alternatively, an inductor (choke) may also be used to increase I/O cable impedance and reduce radiation.

Generally, the shield surrounding low-frequency signals should be grounded at one end, and that for high-frequency signals, at both ends. For example, in Figure 4 where the analog section is grounded, an input cable shield would only be grounded at the analog circuitry end. Shielded cables carrying digital signals (e.g., MC68HC11 SCI data) should be grounded at both ends to ensure that the shield be as close as possible to ground potential throughout its length. If this configuration is not practical, the next best configuration is to ground only the signal source end of the shield. Caution, grounding at both ends of a long cable can cause large power-frequency ground-loop currents to flow due to potential differences between the shield grounding points. This problem can also be removed through filtering or the addition of a common-mode choke or balun (see Reference 3).
PCB LAYOUT GUIDELINES

A successful EMI design starts with good board design. As discussed earlier, the two criteria of most concern are signal-path inductance and loop area. The inductance of a flat conductor (e.g., a PCB track) above a current-return path is as follows:

\[ L = 2 \ln(2 \pi h/w) \text{nH/cm} \]  

where:
- \( h \) = height above current-return path
- \( w \) = track width

Evaluating Equation (4) for a height above a current-return path of 1 mm and a track width of 0.5 mm, \( L = 5.1 \text{nH/cm} \). The relationship is logarithmic, so doubling the track width will not halve the inductance, however, it will make a significant difference and is always worth doing. For example, doubling the track width to 1 mm makes \( L = 3.7 \text{nH/cm} \). A track width of 5 mm makes \( L = 0.5 \text{nH/cm} \), which is of the order required for effective decoupling loops and reducing common-mode radiation problems.

Use of a multilayer PCB will provide low-inductance supplies, though at an additional cost. The recommended arrangement is to place the supply and ground planes on the outside, sandwiching the signal lines between them. This arrangement will also provide some shielding. To minimize crosstalk, signals on adjacent layers should be routed perpendicular to each other wherever possible. If a multilayer board is not used, fill all unused area with ground plane; avoid creating ground loops that can cause EMI problems. For example, a ground loop is discovered and subsequently broken with a small gap. This technique is acceptable at DC, but at high frequencies the gap capacitance may effectively close the loop and create a large loop antenna. Apart from the radiation problems, large ground loops can also make a system more susceptible to malfunction when subjected to an electrostatic discharge (e.g., through a membrane keypad) or other external EMI source.

Reducing loop area through decoupling and careful layout will reduce RFI. The smaller footprint of surface-mount components can be used advantageously in reducing loop area. For PCBs without a ground plane, signal lines should ideally have a ground-return path as close as possible to them to minimize loop area. In the case of address/data lines, this arrangement may be impractical; thus, routing at least one ground-return track adjacent to each of the eight lines and keeping the lines as short as possible is a good compromise. For the address lines, route the ground return next to A0 (in the case of a word-sized bus, A1), since this line is likely to be the most active. Ground and supply loops with long or thin tracks can
be easily identified by tracing them on a printed copy of the PCB artwork using colored marker pens. As previously mentioned, any unused area should be filled with ground plane.

The system clock is often a primary source of radiation. The clock components should be closely grouped, and all clock lines should be as short as possible and have adjacent ground tracks or ground plane. To avoid crosstalk contamination and subsequent radiation problems, the clock circuitry should be located away or shielded from any I/O signal lines or circuitry. For example, mixing clock and I/O buffers in one package is not good practice.

Another source of RFI is an abrupt change of direction of a PCB track which effectively look like impedance discontinuities and will radiate accordingly. For HCMOS designs, it is important to ensure that 90-degree track-direction changes do not occur (see Figure 7).

![Figure 7. Incorrect (a) and Correct (b) PCB Track Layout for HCMOS Designs](image-url)
Finally, all unused inputs to HCMOS devices should be terminated to prevent unintentional random switching and noise generation. Also, unterminated CMOS inputs tend to self-bias into the linear region of operation, which can significantly increase DC current drawn. They are also more susceptible to electrostatic discharge damage.

SIMPLE RFI DIAGNOSTIC TOOL

An article (reprinted by permission of *EMC Technology* magazine, Reference 5) detailing the construction of a set of RFI diagnostic tweezers is included with this application note. After applying the previously discussed techniques to attain EMC, if a radiated EMI problem exists, this simple tool may be used to speed up the identification of potential problem areas on a PCB.

CONCLUSION

EMI control has left the specialized realms of electronic design (e.g., military) and is rapidly becoming an industry-wide phenomenon. Although the application of good system design will always be a prerequisite to achieving EMC, it is reasonable to suppose that similar design concepts could also be applied to the source of most of the radiation, the VLSI HCMOS device. To respond to these and other customer demands for higher performance machines, Motorola is investigating new system and circuit design, layout, and alternative packaging techniques. This research may help to reduce the likelihood of problematic RFI when using HCMOS MCU devices; however, the user's awareness and understanding of the problem will remain the most vital step towards product EMC.

REVIEW OF KEY POINTS

Differential-mode radiation features are as follows:

1. The system clock is often the primary source of radiation. Avoid ground loops and long tracks (always take the most direct route). Wherever possible, clock tracks (or any other signal) should have adjacent ground-return tracks. Minimize the number of devices requiring the system clock. Ensure that clock circuitry and associated lines are located well away or shielded from PCB I/O tracks or circuitry. Never mix clock and bus or I/O drivers in the same package — use separate buffer drivers for clock and buses.
2. Ensure that decoupling capacitors are as close as possible to the device supply pins to reduce the loop area through the capacitor. Always parallel decouple large-value (DC ballast) capacitors with one or more smaller high-frequency capacitors (check their equivalent series inductance (ESL) and maximum frequency rating).

3. In addition to local device decoupling, decouple the power supply where it enters the PCB. A ferrite bead (e.g., Z>50 Ω at 100 MHz) will also help prevent switching transients from going off-board.

4. For PCBs without a ground plane, minimize address/data line loop areas by routing a minimum of one ground-return track adjacent to each of the eight lines and by keeping the lines as short as possible. For the address lines, route the ground return next to A0 because this line is likely to be the most active. Note also that long address lines will ring, which is another potential source of RFI. These lines may need to be individually terminated (see Reference 4). Operating an MCU in single-chip mode will almost eliminate radiation from address/data lines (still exists internally, of course).

5. Avoid ground loops. Remember that breaking a loop with a small gap may be fine at DC but gap capacitance may effectively close the loop at RF frequencies, creating a large loop antenna. Apart from the radiation problems, large ground loops can make a system more susceptible to malfunction when subjected to external EMI sources.

6. Using a printed copy of the PCB artwork and a marker pen, trace the ground and supply tracks. Long, thin, or looped tracks can then be easily identified and subsequently modified.

7. Terminate all unused inputs to prevent unintentional random switching and noise generation (in addition, unterminated CMOS inputs tend to self-bias into the linear region of operation, significantly increasing the DC current drawn).

8. The smaller footprint of surface-mount components may be used advantageously to reduce loop areas.

Common-mode radiation features are as follows:

1. Ensure a good ground plane and choose the external ground connection to minimize the overall common-mode voltage drop (see Reference 4). Increase both supply and return-supply track widths (as a general rule, cover as much as possible of the unused part of a PCB).

2. A grounding scheme that isolates digital and I/O (including any analog sections) reduces radiation from I/O cables. Shielding these cables is ineffective if the shield termination grounds are noisy. In digital systems, the shield should be connected to noise-free grounds at both ends. If this configuration is not possible, then ground only the source end.
3. A choke may be effective in reducing the radiation from an I/O cable. Also available are a variety of other passive RFI filter elements which shunt the common-mode current to ground. The effectiveness of these devices will depend upon the condition of the shunt ground.

REFERENCES


5. EMC Technology and Interference Control News magazine, Gainesville, Va: Don White.
EMI/RFI Diagnostic Tweezer Probes: A Construction Article

by Frank Moriarty
General DataComm, Inc.
Middlebury, CT 06762

The RF bypassing tweezer probes described in this article will give their users the ability to quickly and conveniently locate EMI trouble spots at which a permanent suppression component would likely lower the overall EMI. Equally important, these probes will quickly eliminate those circuit points at which a permanent bypassing fix would not be effective. Though simple (isolated tweezer blades spanned with a capacitive element), their effectiveness greatly reduces EMI troubleshooting time.

This article includes three different models of the basic idea, i.e., a fixed, broadband, non-tuning probe; a tunable model; and a band-switching (three bands) model. The practical performance of each probe was initially tested in both active square wave and coaxial sine wave test circuits. Also, these probes have been used for several years in “on the job” RF suppression work. The time saved has been priceless, and head scratching is held to a minimum.

While the usefulness of these probes is perhaps priceless, the material count and cost will be extremely low—under $5.00—as shown in Table 1. These probes are also available pre-built.

To help in choosing which probe would be best suited to your particular situation, see Table 2. It lists the frequency bandwidths and average attenuation for each of the three types of probes investigated. The Table 2 data was derived from comparing sine wave (50 Ω coaxial) and active circuit probe frequency/attenuation signatures, and it reflects the most conservative data produced from these two test methods. The active circuit chosen, a 5.376 MHz, TTL, square-wave hybrid oscillator, produced an output rich in harmonics. The controlled impedance data (sinewave, 50 Ω coaxial) was, of course, much more predictable. Though different circuits will produce different data, it is felt that Table 2 can be considered a general guide in probe frequency selection and can also be used as an aid in bypass selection.

The base for either of these RF bypassing probes is a simple pair of special tweezers. Tweezers enable quick, one-hand, firm mechanical connection to circuit points. They can be spread to reach circuit points as much as 9 cm (3.5 in) apart, which will cover most applications. Longer tweezers could be substituted for greater spans; however, the increased blade inductance would have to be taken into consideration as the bandwidth parameters would change. The tweezers blades’ inherent low impedance over the frequency range tested (82 nH average) is low enough for them to be of practical use in this application, over the frequency range covered in this article. Figure 1 illustrates the tweezer assembly, and Fig. 2 is a photo of a completed assembly.

**General Construction Steps**

1. The first problem is to split a pair of tweezer blades into two halves. High-quality, spot-welded tweezers are very difficult to split. The lower-quality type, listed in the Table 2 parts list, are very easy to split as they are the unwelded, single-piece, fold-over type. These are quite adequate for this application. If you choose to split the high-quality variety, I recommend drilling

<table>
<thead>
<tr>
<th>Table 1—Parts List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qty.</td>
</tr>
<tr>
<td>1 ea</td>
</tr>
<tr>
<td>A/R</td>
</tr>
</tbody>
</table>

**GENERAL, (all types)**

**Fixed Probes**

| A/R | Capacitors, (Fixed) | C40 | $0.43 |
| 220, 47, 20 10 pF | | series |

**Tunable Probe**

| 1 | Tuning Capacitor | ARCO | Arco 424 | $1.20 |
| (25 tp 150 pF) |

**Bandswitching Probe**

| 1 | Toggle Switch, SPDT, Center Off | Any | $1.95 |
| 1 | Capacitor 220, 39, 20 pF | C40 | $0.43 |

**Table 2—Tweezer Performance Chart**

<table>
<thead>
<tr>
<th>Probe Type</th>
<th>Bypass Element</th>
<th>Span in MHz</th>
<th>Bandwidth in MHz</th>
<th>Average Attenuation, dB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>220 pF</td>
<td>31 - 85</td>
<td>54</td>
<td>17</td>
</tr>
<tr>
<td>Fixed</td>
<td>47 pF</td>
<td>70 - 129</td>
<td>59</td>
<td>13</td>
</tr>
<tr>
<td>Fixed</td>
<td>20 pF</td>
<td>108 - 150</td>
<td>42</td>
<td>11</td>
</tr>
<tr>
<td>Fixed</td>
<td>10 pF</td>
<td>140 - 220</td>
<td>80</td>
<td>11</td>
</tr>
<tr>
<td>Band-Switching (Low Band)</td>
<td>220 pF</td>
<td>27 - 71</td>
<td>44</td>
<td>12</td>
</tr>
<tr>
<td>Band-Switching (Mid Band)</td>
<td>39 pF</td>
<td>68 - 120</td>
<td>52</td>
<td>12</td>
</tr>
<tr>
<td>Band-Switching (High Band)</td>
<td>20 pF</td>
<td>105 - 140</td>
<td>35</td>
<td>11</td>
</tr>
<tr>
<td>Tuning</td>
<td>25/150 pF</td>
<td>30 - 140</td>
<td>110</td>
<td>14</td>
</tr>
</tbody>
</table>

**SUMMARY**

<table>
<thead>
<tr>
<th>Probe</th>
<th>Best BW in MHz**</th>
<th>Best Atten. in dB *</th>
<th>Highest Freq. X</th>
<th>Most Convenient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandsw.</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Tuning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Based on 30 to 140 MHz Span
** Evaluation based on a set of four fixed probes.

NOTE: Tweezer parameters derived from sine, and TTL square wave test circuits, (worst case listed), and are subject to variations from circuit to circuit.
out the spot welds first. After splitting the blades, you may want to shape the tips on a grinding stone. For getting between narrow chip pins, these tips should be thin. Sharp points are also helpful for good probe contact.

2. The next task will be to electrically isolate each tweezer blade from the other and then to bond the two blades together. Sandwich a piece of thin insulator (a piece of wooden ice cream stick 25 mm (1 in) long will suffice) at the upper portion of and between the tweezer halves as shown in Fig. 1. Secure the assembly with alligator clips or shrink tubing. Be sure to keep the tweezer tips parallel. Cover the top 3 mm (0.125 in) portion of the assembly with a small piece of shrink tubing so that you will have an epoxy-free area for soldering the capacitor later on.

3. Liberally coat all sides of the top 33 mm (1.3 in) of your assembly with a generous amount of epoxy. Allow 24 hours hardening time to ensure a good set.

4. Choose the assembly you wish to build; i.e., fixed, tunable or band-switching probe per Table 2. Each probe type has its advantages and disadvantages in bandwidth covered, attenuation characteristics and convenience. The chief advantage of the band-switching and fixed probes are their non-tuning convenience. The chief disadvantage of the fixed probes is the need to have more than one probe to cover a wider frequency range. On the other hand, the tunable probe covers a wide range of frequencies but requires tuning. Overall, the band-switching probe offers the most convenience with a slight loss in attenuation capability. Personally, I have made good use of all three types. Your situation, bandwidth of interest, etc. will be major factors in this decision.

If you choose to build the single-element probe and, depending on the frequency range you wish to cover, you may want to build three or more with different capacitive elements to cover a wider frequency range. Please note that there is an upper frequency limit in the practicality of any of these probes, as covered later in this article.

**Fixed-Element Probe, Final Assembly**

1. Referring to Fig. 1 as a general guideline, choose the capacitor element you wish to solder to the tweezer blades. Note that the upper frequency limit is roughly 300 MHz here, due to hand capacitance, the basic tweezer capacitance of 10 pF average and the tweezer blades' inductance of 82 nH average. The author limited testing and use to 220 MHz.
2. Solder the capacitor across the top of each of the tweezer halves as in Fig. 1. Keep capacitor lead lengths short and use heat sinks on them.

Note: Soldering to stainless steel tweezers requires a special flux. I tried several fluxes, including ammonia, with no success. If you do not have the proper flux available, then try a method I call "electro flux." Figure 3 shows a schematic for the electrical part of this method. With this hookup, alternately charge and then discharge the large electrolytic capacitor, through a length of solder, to the top left corners of each tweezer half. The arcing between the solder element and the tweezer will eventually cause a solder buildup at this point. This buildup will not be readily obvious, but it will be quite adequate when tinned. For tinning, use a good electronic grade of rosin core solder. A paste flux would also aid in this tinning.

3. Decide on the desired spread of the tweezer blades. For instance, if the largest spread you will require is for a 50 mm (2 in) gap. Caution: when bending the tweezer blades, avoid putting excessive stress on the epoxy bond.

4. Finish off the probe with shrink tubing on each blade and on the top portion of the tweezer assembly to help reduce hand capacitance effects and possible PC card shorts (should you happen to drop the probe into a circuit card). Note: this top covering could be a removable type which would enable the removal and soldering of different capacitors for experimental purposes or to avoid building two or more probes to cover wider frequency ranges.

**Testing Your Probe**

1. Make the 50 Ω coaxial test setup shown in Fig. 4.

2. Connect your probe across the open end of the coaxial "Tee" connector.

3. Your readings, being in a controlled impedance test circuit, will only approximate the data listed in Table 2, which was derived by comparing data from differing circuits and indicates the worst-case effects of that comparison.

4. Of course, if you are testing the tunable probe, you will have to tune for each frequency being tested to get maximum attenuation.

**Use of the Single-Element Probe**

As a typical use of the probe would be in chasing down radiated EMI signals, the following will cover that aspect of its use. Adjustments can be made for other situations.

**Caution:** Do not use these probes in circuits containing voltages in excess of 50 Vdc.

1. Begin by tuning in the EMI signal in question. Determine its exact frequency. A schematic analysis of various EMI harmonic frequencies or circuit probing with a "sniffer probe" (see Table 1) will determine the location of suspected circuit points which might be generating the offending frequencies.

2. Assuming a typical radiated test setup using an antenna for a pickup device and a receiver/analyzer to observe the EMI, tune in the EMI signal to be investigated and use your tweezer probe to probe the suspected circuit "hot spots" as determined in step 1, above.

   Note: As you are using a bypass device, one blade of the tweezer must be in firm contact with a circuit common point, and the other blade should be in contact with the circuit point under investigation. The best common point is generally the common pin on the chip being probed.

   Should probing show little or no decrease in the observed EMI at a particular circuit point, or be seen to increase the EMI at this point, is generally an indication that this circuit point would not yield to a successful, permanent EMI bypass fix. However, the points which indicate a small decrease (3 or 4 dB) may be part of a dual EMI problem, i.e., a circuit which has two or more problem areas. Keep these points in mind if later probing is inconclusive. Such a condition may require two or more permanent fixes. In any event, with experience your probing analysis will keep time-consuming "cut and try" methods to a minimum.

   Any circuit point which indicates a moderate or large decrease due to probing should be noted as a likely point for a permanent EMI bypass fix. As will all things, experience will bring more knowledge. As you use your probes more often, you will be able to quickly determine those areas at which a permanent fix would be successful. Also, you will be able to more accurately choose the type and size of fix to use, i.e., a series or bypassing fix. Other types of EMI fixes which might be required, such as shielding or grounding improvements, will also be determined by these methods.

**Tuning Probe**

1. Refer to the previous "general construction" and applicable "fixed probe" paragraphs to create a completed tweezer assembly, minus the capacitive element. One exception here would be to apply the shrink tubing on the tweezer blades before joining them. Shrink tubing should stop approximately 25 mm (1 in) from the top of each blade for bonding purposes.

2. Carefully epoxy the variable capacitor to a tweezer blade. Be sure the variable capacitor's active parts do not come in contact with either of the tweezer blades. Solder the variable capacitor to the upper half of the tweezer blades. Solder one element of the variable capacitor directly to a tweezer blade. The other half of the capacitor element will have to be soldered through a short length of wire. Keep this wire as short as possible to hold your usable bandwidth as wide as possible.

   It is good practice to have the movable portion of the variable capacitor (rotor) as the circuit ground connection. The rotor part of the tweezer assembly can easily be identified by stripping back a longer piece of the shrink tubing on the tip porton of the rotor's tweezer blade.

3. Carefully epoxy the variable capacitor to a tweezer blade. Be sure not to get epoxy on any of the capacitor's moving parts. Allow ample hardening time, then cover the assembly with shrink tubing. Be sure the shrink tubing allows for free movement of the capacitor's moving parts. The minimum movement of the assembly must allow for three full turns on the variable capacitor's adjustment screw.

**Using the Tuning Probe**

1. With your radiated test setup intact, place a "Tee" coaxial adapter at the input of the receiver/analyzer.

2. Tune in the EMI signal to be investigated.

3. Connect your tuning probe across the open end of the coaxial "Tee" and tune the variable capacitor for a minimum signal as observed on your receiver. For maximum attenuation due to circuit detuning, an optional, additional in-circuit tweak might be desirable.

   Note: Resonant tuning begins at approximately 60 MHz and ends around 130 MHz. Frequencies below 50 MHz are probed with the adjusting screw fully clockwise (tight). Likewise, fre-
frequencies above 130 MHz are probed with minimum capacitance (three turns counterclockwise on the adjustment screw). The frequencies from 60 to 130 MHz should be tuned to resonance (minimum amplitude as seen on your receiver/analyzer). Of course, different assemblies will vary somewhat from these parameters.

Your tuning probe is now tuned for optimum attenuation at the EMF frequency under investigation.

4. Refer to the preceding section, “Using the Single-Element Probe,” for a guide to probing.

**Band-Switching Probe**

The band-switching probe was the next obvious step in the evolution of these probes. It performs quite well and is very convenient to use.

1. Proceed with the basic construction steps as outlined in the “general construction” paragraphs, with the exception of leaving an epoxy-free area for mounting the band switch and applying the shrink tubing on the tweezer blades before bonding.

2. Mount, secure and epoxy the band switch to one of the tweezer blades (see Fig. 1). Allow ample hardening time for the epoxy.

3. Referring to Fig. 5, mount the three capacitors as shown and wire the band switch. Use the capacitor leads for wiring, keep lead lengths short and apply heat sinks to the capacitors before soldering. Note that capacitor logic is single capacitor, series, hand series parallel (moving toggle from left to right). This method was chosen to obtain closer overlapping bands and to take advantage of the band switch’s terminals for mounting the capacitors.

4. Cover the capacitors and exposed band switch terminals with heat shrink tubing.

**Using the Band-Switching Probe**

Use this probe as with the fixed probes, with the exception of the band-switching feature. If wired as shown in Fig. 5 (pictorial view), and if you are using a typical toggle switch, the low band will be with the toggle left of center, the high band will be with the toggle in mid position (off), and the mid band will be with the toggle in the far-right position.

Note: If it is desired to know the actual capacitance involved in a particular fixed-probe effect, simply add 15 pF (residual capacitance of the tweezers when constructed as per Fig. 1) to the probe’s capacitive element value. The capacitance of the tuning probe, however, would have to be measured. For the band-switching probe, being series and series parallel wired, add the 15 pF to the following:

<table>
<thead>
<tr>
<th>Band</th>
<th>Capacitance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>220 pF</td>
</tr>
<tr>
<td>Mid</td>
<td>47 pF</td>
</tr>
<tr>
<td>High</td>
<td>19 pF</td>
</tr>
</tbody>
</table>

This information is useful when looking for a minimum capacitance for a permanent bypass fix when working with a circuit which cannot drive a large capacitance without unacceptable distortion of the wave shape.

**Figure 5—Bandswitching Wiring**

Note: the probes discussed in this article are also available in pre-built form. For a price list and ordering information, contact F&M Electronics, 41 Chestnut St., Seymour, CT 06483, (203) 888-4847.
M6805 16-Bit Support Macros

If your microcontroller (MCU) application requires a small amount of program memory and not much raw computing power, the M6805 MCU Family is a most logical choice, given their low cost. But do not cross the M6805 Family off your selection list just because you need some 16-bit indexing and/or 16-bit operations, such as the higher cost M68HC11 Family provides. While the 8-bit x index register of the M6805 Family cannot access the entire M6805 12/13-bit address space and its single 8-bit accumulator cannot directly do 16-bit operations, advanced software techniques can be employed to work around the limitations of the M6805 Family. This application note gives specific details and examples of these techniques.

The code samples given here are available in source code form on the Motorola FREEWARE Bulletin Board Service (BBS), by telephoning 512/891-FREE (512/891-3733). The FREEWARE line operates continuously (except for maintenance) at 300-2400 baud, 8 data bits, 1 stop bit, and no parity. Sample test files are also included. Download the archive file, MACROS05.ARC, to get all the files. All files are suitable for use with the Motorola Development Systems M6805 Portable Assembler for MS-DOS, known as PASM05. Other assemblers may also be supported, but consult the BBS for details.

The techniques used here involve a combination of macros and RAM-based subroutines which use instruction modification. Macros allow programming on a higher level of thought with less chance for introducing errors.

Instruction modification is a technique of altering an instruction just prior to its execution. The modification requires that the instruction be in RAM and can involve the opcode and/or the operand portion of the instruction. By determining the exact instruction/operand needed in advance of execution, greater efficiency in execution speed and code size can be obtained. There is a significant risk in using the instruction modification technique because if used improperly, the program will either fail to work properly at best, or crash at worst. When the technique is used, great care must be exercised to ensure correct operation in all possible cases.

To illustrate the instruction modification technique, consider the following instruction at location $0050 in RAM memory.

0050 C6 04FF LDA $4FF

The LDA instruction consists of three bytes: an opcode byte ($C6) and two bytes that hold the extended address ($04FF) of the operand. When executed, the A accumulator will be loaded with the contents of location $04FF. Now consider what happens when the following instructions at location $870 are executed and how the previous memory locations are changed.

0870 A6 05 LDA #$05
0872 B7 51 STA #$51
0874 A6 2C LDA #$2C
0877 B7 52 STA #$52

These instructions store $05 into location $0051 and $2C into location $0052, which is the operand address of the opcode byte at location $0050. This has the effect of changing the instruction at location $0050 to the following:

0050 C6 052C LDA #$52

When location $0050 is executed, the A accumulator will now be loaded with the contents of location $052C, i.e., the instruction at location $0050 has been modified! Note that the original source listing will only show "LDA $4FF", as the instruction is only changed at execution time, not assembly time.

When instruction modification is used in a ROM-based system, the RAM code must be initialized (from the ROM) before being used. This can be as simple as a few LOAD/STORE instructions for a small routine, or a MOVE BLOCK routine may be required for larger routines.

As with all things of value, there is a price to be paid. The price for using these macros is rather small, namely, 23 bytes (16 for RAM subroutines and 7 for local storage) of direct addressing memory, i.e., in the range of $0-$00FF. The macros have a small size/execuion speed penalty associated with them that varies from zero to 50 percent, depending on the frequency and type of macros used. An estimated typical cost for an entire program with moderate macro usage would be in the 5 to 20 percent range. But this is small cost to pay for error-free code generation in areas of the program where speed is not critical. By judicious usage/choice of macros, the cost can be held closer to the 5 percent range.

Part of this cost is associated with the structured code technique of preserving registers and another part is involved with setting up the proper condition codes. The rest of the cost is inherent in the fact that M6805 MCUs must do extra work in software to simulate the hardware capabilities built in M68HC11 MCUs. Because these macros have been fine-tuned for size and speed efficiencies, the overhead cost of register preservation is typically 4 bytes and the overhead cost of condition code setting is typically 2 to 4 bytes, per macro invocation. It can be as high as 200 percent, as is the case of the MOV B macro for extended addressing operands (16 versus 8 bytes if no register preservation/condition code initialization is done) or as low as zero percent for direct addressing operands.

423
The DREG macros have almost no overhead associated with them since the DREG is implemented using the already existing A and X registers. The overhead price for DREG macros is only 2 bytes of local storage (TEMPAS, TESTAS), because the RAM subroutines are only needed for XREG and YREG support. The Add and Subtract DREG macros (ADDD, SUBD) have the most overhead because they must save and restore a working register (A), but even this is rather minimal. If only DREG macros are used, it is estimated that the code size could expand from zero to 5 percent over straight assembly language, depending on how many and which type of DREG macros are used.

The XREG and YREG macros have the most overhead, since they are implemented using RAM memory locations. The Load and Store via XREG/YREG macros (LDAXY, STAXY) have the most overhead when nonzero offset values are used (26 bytes versus 2 bytes for zero offset, or 1300 percent). Thus, nonzero offset usages should be avoided unless absolutely necessary. There is also some inefficiency associated with internally maintaining two copies of each register, but it actually helps in the overall implementation.

The Increment, Decrement, and Move macros (INC.B, INC.W, DEC.B, DEC.W, MOV.B, MOVE.W, MOVE) have zero overhead. The DEC.B and INC.B macros have zero overhead when used with direct addressing operands, while the INC.W and DEC.W macros have high overhead due to setting the proper condition code based on the resultant 16-bit value.

To use these 16-bit macros, here is a quick summary of the steps involved for use on an IBM-PC with the Development Systems PASM05 macro assembler, which should already have been installed per the instructions supplied with the product. The sample MS-DOS commands are shown in upper case for clarity only (except as noted), as MS-DOS accepts either upper or lower case.

1. Create a new directory for your project and change directory to it as shown below.
   
   C>mkdir proj
   C>cd proj

2. Download the archive file, MACROS05.ARC, from the Motorola FREEWARE Bulletin Board to your project directory and then de-archive it as shown below. See the FREEWARE bulletin file, archive.bul, for de-archiving information.
   
   C>pkarc  macros05. arc

3. Read the README file first, and then read the Notes header in the MACROS05.MAC and RAMSBR.INI files. In the MACROS05.MAC file, study the individual macro headers of the macros you intend to use, especially the examples shown.

4. Write your source code using the example shown in the Notes header of the MACROS05.MAC file, as illustrated in Listing 1. lines 1030-1460. If the two ORG statements, lines 1130 and 1180, are not valid for your application's memory map, change them to the appropriate values. Notice especially how the initialization of the RAM subroutines is accomplished by the MOVE macro in line 1200 using the ROM code generated by the INCLUDE statement in line 1400. Or you can expand the BBS EXAMPLE.S file into your source file by first making a copy of it and then editing the copy as shown below (PE is the IBM Personal Editor command, but you can use any editor you feel comfortable with).
   
   C>copy example.s myfile.s
   C>pe myfile.s

5. Invoke PASM05 to assemble your new source file as shown below. Because the options for PASM05 are case sensitive, be sure to enter them just as shown.
   
   C>pasm05 -eq -1 myfile.lst myfile.s

   This produces listing file MYFILE.LST and a COFF relocatable object file MYFILE.O.

6. To produce an absolute S-record object file suitable for programming an EPROM, the COFF relocatable object file must first be linked to an absolute object file (MYFILE.OUT) and then converted to an S-record file (MYFILE.MX), which many commercial EPROM programmers, such as Data I/O, recognize. Enter the commands shown below to create the S-record file.
   
   C>pld -o myfile.out myfile.o
   C>ubuilds myfile.out

   Steps 5 and 6 can be simplified by copying and editing the BBS batch file, MAKE1.BAT, in a manner similar to that shown in Step 4, except here we want the new batch file, MAKE.BAT, to process MYFILE instead of TEST1. The resultant MAKE.BAT file should be similar to the text shown below. The two DEL commands at the end are optional, as these files are no longer needed.
   
   pasm05 -eq -1 myfile.lst myfile.s
   pld -o myfile.out myfile.o
   ubuilds myfile.out
   del myfile.o
   del myfile.out

   To accomplish Steps 5 and 6, invoke the batch file MAKE.BAT by simply typing its name as shown below. MS-DOS will execute each line of the file as if you had typed it on the keyboard.
   
   C>make
7. Consult your EPROM programmer’s user manual for how to load the resultant S-record object file (MYFILE.MX) and program your chosen device.

Because of the length of the source listings involved and because these listings are intended to be self-explanatory, the rest of this discussion will only deal with salient points which might need clarification for the reader. Line numbers have been added at the beginning of each line for identification purposes in this Application Note, i.e., they are not present in the actual source file.

These macros have been written so that any syntax error will result in failing to the end of the macro where the FAIL directive will force a “Macro syntax error detected” message. Proper usage results in exiting the macro early via the MEXIT directive and thus avoids the FAIL directive at the end.

The file header of Listing 1 (lines 20-1800) gives specific details of the macros supported and their general operation and restrictions. Line 1810 defines where a seven byte block of low memory will be allocated for support of these macros (see lines 20430-20490). Line 1830 disables the output listing to avoid repetitious output, while line 19630 re-enables the listing.

A lot of conditionals (IFxx) involve testing to see if direct addressing can be used, as it is more efficient (one less byte and one less execution cycle per instruction) than extended addressing, as illustrated by lines 2090-2210. Also, conditionals check to see if shorter instruction forms can be used, like ‘CLRA’ instead of ‘LDA #0’, as in these same lines. For this same reason, it is most efficient to place the RAM subroutines, and thus the XREG and YREG, into direct addressing space ($0000-$00BF). Remember that the M6805 uses a fixed stack area in direct addressing space of $00C0-$00FF.

The COMPARE macros (CPD, CPXR, CPYR) use a shortened form when immediate addressing is specified and either of the halves is zero, i.e., no “CMP #0” instruction is generated. Lines 3480-3660 are typical of this technique. As always, testing for zero is most efficient in computer architecture. The 16-BIT INDEXING macros (LDAXY, STAXY) are most efficient when used with a zero offset, as typified in lines 3980-4060, but will function with any sized offset (lines 4070-4210). Nonzero offsets must first be added to the indexing register (XREG or YREG), the operation performed via the appropriate RAM subroutine, and lastly, the indexing register must be restored to its original value.

Lines 19640-20410 comprise the RAM subroutines which are the keystone of the macros using the XREG and YREG pseudo 16-bit index registers. It is here that the XREG and YREG pseudo registers are defined and stored as the second and third bytes of extended addressing LDA and STA instructions using EQU directives (lines 19910-19920, 20060-20070, 20210-20220, and 20360-20370). Instructions that store values to XREG or YREG will thus modify the instruction’s effective address, hence the name instruction modification. These RAM subroutines must be initialized before they can be used, and so a mirror image of this code with altered labels is provided in the RAMSBR.INI file (as seen in Listing 2). This file can then be simply INCLUDED in the ROM section of the user’s code and copied to RAM via the MOVE macro, as discussed in lines 19660-19760 of Listing 1.

Listing 2 is the RAM subroutine initialization file and is essentially a copy of the RAM subroutines defined in lines 19640-20410 of Listing 1, but the labels have all been prefixed with a period (.). This allows using the MOVE macro to copy this RAM initialization code from a ROM to RAM (see lines 370-410).

Lines 1070-1090 verify that the number of RAM subroutine initialization bytes is identical.
Listing 1 – MACROS05.MAC File

PAGE

**********************************************************
*
macros05.mnc 1.0
*
**********************************************************

Module Name: MACROS05 - M6805 Macros

Description:

This file contains macros and subroutines to support pseudo-registers on the 6805 that simulate registers and addressing modes available on the 68HC11. It is suitable for "black box" operation, i.e., the macros may be used without knowledge of how they work. A list of the supported macros follows. Consult the individual macro headers for usage details and see the "Notes" below.

LDD Load DREG
STD Store DREG
ADDR Add DREG
SBD Add DREG
CFD Compare DREG
LDAXY Load A via 16-bit pseudo-register (XREG or YREG)
STAXY Store A via 16-bit pseudo-register (XREG or YREG)
LDXR Load XREG
STXR Store XREG
INCR Increment XREG
DECR Decrement XREG
CPXR Compare XREG
LDXR Load YREG
STXR Store YREG
INCYR Increment YREG
DECYR Decrement YREG
CPYR Compare YREG
DEC.B Decrement byte
DEC.W Decrement word
INC.B Increment byte
INC.W Increment word
MOV.B Move byte
MOV.W Move word
MOVE Move block of memory

General Information:

The following pseudo-registers are supported.

DREG = pseudo 16-bit accumulator (A,X registers, A is MS half)
XREG = pseudo 16-bit index register
YREG = pseudo 16-bit index register

The following terms are used.

# = specifies immediate addressing mode
<address> = address/value operand (absolute or immediate)
<offset> = unsigned 16-bit offset for indexed addressing

Notes:

1. Motorola reserves the right to make changes to this file.
2. Although this file has been carefully reviewed and is believed to be reliable, Motorola does not assume any liability arising out of its use. This code may be freely used and/or modified at no cost or obligation by the user.
3. This file was made for use with the Motorola Development Systems MC6805 Portable Assembler/Linker for MS-DOS, known as PASM05 and PLD, as released on 82HCVBASM B* and 82HCVBLINK B*. Consult the PASM and PLD reference manuals, part numbers M68HASM/D1 and M68HLINK/D1 for more details.

426
3. These macros were made for ABSOLUTE assemblies only, i.e., for use with ORG directives. While most of the macro concepts will work in relocatable assemblies (BSCT, DSCT, PSCT, ASCT, XDEF, and XREF), errors will be generated because PASM limits the use of external symbols in expressions and because the value of an expression must be known at assembly time for IFxx directives to assemble the proper code. The first restriction is a result of limitations in the COFF object file format. If it is desired to have these macros work with relocatable assemblies, modifications similar to below should be made, but be forewarned of the increased inefficiencies in size and speed. Consider the following code to change the LDD macro so that an XREF parameter, \$, can be loaded as an immediate value.

```
 00760  * LDA  \.8
 00770  * LDX  \.8+1
 00780  * BRA  \.9
 00790  * \.8  FDB  \1
 00800  * EQU *
```

4. In order to efficiently support both LOAD and STORE operations for the pseudo 16-bit index registers, there are actually two such "registers", i.e., one for LOAD and one for STORE as defined below. These routines maintain both "registers" with the same value, and so the programmer may think of them as one register.

```
 00870  * XREG1$ = 16-bit XREG for LOAD operations
 00880  * XREG2$ = 16-bit XREG for STORE operations
 00890  * YREG1$ = 16-bit YREG for LOAD operations
 00900  * YREG2$ = 16-bit YREG for STORE operations
```

5. These macros can be used to create in-line code (speed efficient) or they be placed in a subroutine (byte efficient).

6. Instruction modified code is used here and is denoted by use of the unique string "0-0" (RAM subroutines).

7. Some macros use temporary storage locations (TEMPS$, TESTA$, etc.), so these macros should not be used in any interrupt routine in order to avoid corrupted values!

8. The user must ensure that the code is appropriately placed in the target M680S's memory map, i.e., the RAM subroutines MUST be located in RAM but must not overlap the stack area ($00C0-00FF) unless it can be GUARANTEED there is no conflict! See LOGMEM below to set low memory data storage area!

9. To use this file, either use an INCLUDE statement or just merge this file into your source code file. Consult your assembler's user's manual for the details specific to your situation. When using a ROM controlled system, the MOVE macro should be used to copy the RAM subroutines from ROM to RAM (see the comments after where RAMSBR$ is defined below and note the INCLUDE statement for the RAMSBR.INI file).

Reference the code segment example below for usage ideas (shown in PASM05 for MS-DOS syntax).
01270 * CPD  
01280 * BEQ  MATCH
01290 *
01300 * LDXR  
01310 * LDYR  
01320 * LOOP LDAXY TABLE, XREG
01330 * STAXY RTABLE, YREG
01340 * INCXR
01350 * INCYR
01360 * CPY  #.5
01370 * BNE LOOP
01380 *
01390 * INCLUDE RAMSBR.INI
01400 * TABLE FCB 1, 2, 3, 4, 5
01420 * ADJUST FDB 150
01430 * COST FDB 859
01440 *
01450 *
01460 * END
01470 *
01480 * 10. To assemble, use the following sample invocation lines:
01490 * pasm05 -eq -1 filename.lst filename.s (debug= expanded)
01500 * or
01510 * pasm05 -bf -1 filename.lst filename.s (black box)
01520 *
01540 * 11. Notations used by PASM05 are as follows:
01550 * OPERATORS: Special two character operators used are...
01560 * 1. Logical AND
01570 * 2. Shift Right (0 fill on left) !>
01570 * MACROS: Special notations used are...
01580 * 1. Parameters are positionally named using \0, \1, \2, etc.
01590 * 2. Labels within macros are designated via \a, where "a" is an alphanumeric character. The
01600 * assembler generates a unique label to avoid
01610 * multiply defined label problems.
01620 * 12. Some macros access 16-bit values as LS byte then MS byte in order
01640 * to be more efficient for condition code (CC) setting. This is
01650 * the reverse order that the 68HC11 would access the two byte
01660 * halves. This difference would only be a concern in accessing
01670 * hardware registers, as normal RAM makes no difference. Those
01680 * macros with this difference have an entry in their Notes section.
01690 * 13. The latest version of this file is maintained on the Motorola
01700 * FREeware Bulletin Board, 512/891-FREE (512/891-3733). It operates
01710 * continuously (except for maintenance) at 1200-2400 baud, 8 bits,
01720 * no parity. Sample test files for PASM05 are also included.
01730 * Download the archive file, MACROS05.ARC, to get all the files.
01740 *
01750 *******************************************************
01760 * REVISION HISTORY  (add new changes to top):
01770 *
01780 * 05/16/90  P.S. Gilmour
01790 * 1. Created for Application Note AN1055.
01800 *******************************************************
01810 LO$MEM EQU $0000-7 Low memory ($0000-00FF) storage (7 bytes)
01820

428
LDD = load DREG

Examples:
1. "LDD #START" puts the value of symbol 'START' into the DREG (=A,X).
2. "LDD START" puts the contents of location 'START' and 'START'+1 into the DREG (=A,X).

Register Usage:
A,X = loaded with new value (DREG).
CC = reflects MS half (=A).
All other registers preserved.

Notes:
1. Byte access order is LS, then MS (reversed from 68HC11).

STD = store DREG

Examples:
1. "STD START" stores the DREG (=A,X) into locations 'START' and 'START'+1.

Register Usage:
CC = reflects MS half (=A).
All other registers preserved.

Notes:
1. Byte access order is LS, then MS (reversed from 68HC11).
ADDD = add DREG

** Examples: **
1. "ADDD [#,START]" adds the value of symbol 'START' to the DREG (=A,X).
2. "ADDD START" adds the contents of location 'START' and 'START'+1 to the DREG (=A,X).

** Register Usage:**
- A,X = contains new value (DREG).
- CC = reflects MS half (=A).
- All other registers preserved.

** ADDD MACR **

IFEQ NARG-1
STA TEMPA$
TXA
ADD \(0\)+1
TAX
LDA TEMPA$
ADC \(0\)
MEXIT
ENDC

IFEQ NARG-2
IFC '\0', '#'
STA TEMPA$
TXA
ADD #\(1\)!$.FF
TAX
LDA TEMPA$
ADC #\(1\)!>8
MEXIT
MEXIT
ENDC
ENDC

FAIL Macro syntax error detected!
ENDM

** SUBD = add DREG **

** Examples: **
1. "SUBD [#,START]" subtracts the value of symbol 'START' from the DREG (=A,X).
2. "SUBD START" subtracts the contents of location 'START' and 'START'+1 from the DREG (=A,X).

** Register Usage:**
- A,X = contains new value (DREG).
- CC = reflects MS half (=A).
- All other registers preserved.

** SUBD MACR **

IFEQ NARG-1
STA TEMPA$
TXA
SUB \(0\)+1
TAX
LDA TEMPA$
SBC \(0\)
MEXIT
MEXIT
ENDC
IFEQ NARG-2
IFEQ NARG-1

Examples:
1. "CPD #, BLOCKSZ" compares the value of symbol 'BLOCKSZ'
2. "CPD START" compares the contents of location
   'START' and 'START'+1 with the DREG.

Register Usage:
- CC = reflects DREG comparison (Z-bit only).
- All other registers preserved.

CPD MACR
IFEQ NARG-1
IFEQ NARG-2

FAIL Macro syntax error detected!
**LDAXY** = load A via 16-bit pseudo-register (XREG or YREG)

**Examples:**

1. "LDAXY 0,XREG" loads the contents of the memory location specified by XREG+0 into the A accumulator.
2. "LDAXY ,XREG" loads the contents of the memory location specified by XREG+0 into the accumulator.
3. "LDAXY TBL,XREG" loads the contents of the memory location specified by XREG+'TBL' into the A accumulator.
4. Above examples can be repeated with substituting YREG for XREG.

**Register Usage:**

- **A** = loaded with new value.
- **DREG** = destroyed.
- **CC** = reflects value loaded.
- **All other registers preserved.**

**Default offset = 0**

- **Set nREG= offset + nREG**
- **Restore nREG**

**FAIL** Macro syntax error detected!
* STAXY = store A via 16-bit pseudo-register (XREG or YREG)

* Examples:
  1. "STAXY 0,XREG"  stores the accumulator (=A) into the memory
     location specified by XREG+0.
  2. "STAXY ,XREG"  stores the accumulator (=A) into the memory
     location specified by XREG+0.
  3. "STAXY TBL,XREG" stores the accumulator (=A) into the memory
     specified by XREG+'TBL'.
  4. Above examples can be repeated with substituting YREG for XREG.

* Register Usage:
  CC = reflects value stored.
  All other registers preserved.

* STAXY MACR

* IFNC '\1', 'XREG'

* IFNC '\1', 'YREG'

* FAIL Macro syntax error detected!

* ENDC

* ENDC

* IFC '\0', ''

* JSR STA\1  Default offset= 0

* MEXIT

* ENDC

* ENDC

* IFC '\0', ''

* JSR STA\1  Offset= 0

* MEXIT

* ENDC

* IFC '\0'

* STA TEMPA$

* LDA \12$+1  Set nREG= offset + nREG

* ADD #(\0)!.$FF

* STA \12$+1

* LDA \12$

* ADC #(\0)!>8

* STA \12$

* LDA TEMPA$

* JSR STA\1  Offset= 0

* LDA \11$  Restore nREG

* STA \12$

* LDA \11$+1

* STA \12$+1

* LDA TEMPA$

* MEXIT

* ENDC

* ENDC

* FAIL Macro syntax error detected!
04820  ************************************************************
04830  * LDXR = load XREG                                       *
04840  * LDXR [#, ]<address>                                   *
04850  *                                                     *
04860  * Examples:                                            *
04870  * 1. "LDXR #,START" puts the value of symbol 'START' into the *
04880  *     XREG.                                             *
04890  * 2. "LDXR START" puts the contents of location 'START' and  *
04900  *     'START'+1 into the XREG.                            *
04910  *                                                     *
04920  * Register Usage:                                       *
04930  * CC = reflects MS half (=A).                           *
04940  * All other registers preserved.                       *
04950  *                                                     *
04960  * Notes:                                               *
04970  * 1. Byte access order is LS, then MS (reversed from 68HC11). *
04980  *                                                     *
04990  LOXR MACR                                             *
05000  IFEQ NARG-1                                          *
05010  STA TEMPA$                                           *
05020  LDA (\0)+1                                           *
05030  STA XREG1$+1                                        *
05040  STA XREG2$+1                                        *
05050  LDA (\0)                                            *
05060  STA XREG1$                                          *
05070  STA XREG2$                                          *
05080  IFEQ XREG1$!, $FF00                                 *
05090  LDA TEMPA$                                          *
05100  TST XREG1$                                          *
05110  ENDC                                                *
05120  IFNE XREG1$!, $FF00                                 *
05130  STA TESTA$                                          *
05140  LDA TEMPA$                                          *
05150  TST TESTA$                                          *
05160  ENDC                                                *
05170  MEXIT                                               *
05180  ENDC                                                *
05190  IFEQ NARG-2                                         *
05200  IFC '\0','\#'                                      *
05210  IFEQ XREG1$!, $FF00                                 ! XREG in low memory? *
05220  IFEQ \1                                           ! \0 value?      *
05230  CLR XREG1$+1                                       *
05240  CLR XREG2$+1                                       *
05250  CLR XREG1$                                         *
05260  CLR XREG2$                                         *
05270  MEXIT                                               *
05280  ENDC                                                *
05290  IFNE \1                                           ! not \0 value? *
05300  STA TEMPA$                                          *
05310  IFEQ (\1)!.$FF                                     *
05320  CLR XREG1$+1                                       *
05330  CLR XREG2$+1                                       *
05340  ENDC                                                *
05350  IFNE (\1)!.$FF                                     *
05360  LDA #(\1)!.$FF                                     *
05370  STA XREG1$+1                                       *
05380  STA XREG2$+1                                       *
05390  ENDC                                                *
05400  IFEQ (\1)!>8                                       *
05410  CLR XREG1$                                          *
05420  CLR XREG2$                                          *
05430  ENDC
IFNE (\1)'>8
LDA #(\1)'>8
STA XREG1$
STA XREG2$
ENDC
LDA TEMPA$
TST XREG1$
MEXIT
ENDC
ENOC
05540 IFNE XREG1$.$FF0 ! XREG in high memory?
05550 IFEQ \1 ! #0 value?
STA TEMPA$
CLRA
STA XREG1$+1
STA XREG2$+1
CLR TESTA$
LDA TEMPA$
TST TESTA$
MEXIT
ENDC
05560 IFNE \1 ! not #0 value?
STA TEMPA$
IFEQ (\1)'.$FF
CLRA
ENDC
05570 IFNE (\1)'.$FF
LDA #(\1)'.$FF
ENDC
05580 STA XREG1$+1
STA XREG2$+1
IFEQ (\1)'>8
CLRA
ENDC
05590 IFNE (\1)'>8
LDA #(\1)'>8
ENDC
05600 STA XREG1$
STA XREG2$
STA TESTA$
LDA TEMPA$
TST TESTA$
MEXIT
ENDC
05610 ENDC
05620 FAIL Macro syntax error detected!
ENDM
05630 435
05960  ******************************************************************************
05970 * STXR = store XREG
05980 * STXR <address>
05990 *
06000 * Examples:
06010 * 1. "STXR START" stores the XREG into locations 'START' and
06020 * 'START'+1.
06030 *
06040 * Register Usage:
06050 * CC = reflects MS half (=A),
06060 * All other registers preserved.
06070 *
06080 * Notes:
06090 * 1. Byte access order is LS, then MS (reversed from 68HC11).
06100 *
06110 STXR MACR
06120 STA TEMPA$  
06130 LDA XREG1$+1
06140 STA (\0)+1
06150 LDA XREG1$  
06160 STA (\0)  
06170 IFEQ XREG1$!.SFF00
06180 LDA TEMPA$  
06190 TST XREG1$  
06200 ENDC
06210 IFEQ XREG1$!.SFF00  
06220 STA TESTA$  
06230 LDA TEMPA$  
06240 TST TESTA$  
06250 ENDC
06260 ENDM
06270
06280  ******************************************************************************
06290 * INCXR = increment XREG
06300 * INCXR [[#,]<address>]  
06310 *
06320 * Examples:
06330 * 1. "INCXR" adds one (1) to the XREG.
06340 * 2. "INCXR #,START" adds the value of symbol 'START' to the
06350 * XREG.
06360 * 3. "INCXR START" adds the contents of location 'START' and
06370 * 'START'+1 to the XREG.
06380 * 4. "INCXR ! comment" adds one (1) to the XREG (comment present!).
06390 *
06400 * Register Usage:
06410 * CC = reflects value incremented (Z-bit only).
06420 * All other registers preserved.
06430 *
06440 * Notes:
06450 * 1. Explicit comment character (!) MUST be used when comment field is
06460 * present to prevent confusion with parameters!
06470 * 2. Assumes XREG1$ = XREG2$.
06480 * 3. When parameters are present, this macro becomes "ADD to XREG"
06490 *
06500 INCXR MACR
06510 IFEQ NARG  
06520 IFEQ XREG1$!.SFF00  
06530 INC XREG1$+1
06540 INC XREG2$+1
06550 BNE \\0.
06560 INC XREG1$  
06570 INC XREG2$  
06580 \0. EQU *
06590 MEXIT
06600 ENDC
FAIL Macro syntax error detected!
**DECX** = decrement XREG

**Examples:**

1. "DECX" subtracts one (1) from the XREG.

2. "DECX #,START" subtracts the value of symbol 'START' from the XREG.

3. "DECX START" subtracts the contents of location 'START' and 'START'+1 from the XREG.

4. "DECX ! comment" subtract one from the XREG (comment present!).

**Register Usage:**

- CC = reflects value decremented (Z-bit only).

- All other registers preserved.

**Notes:**

1. Explicit comment character (!) MUST be used when comment field is present.

2. Assumes XREG1$ = XREG2$.

3. When parameters are present, this macro becomes "SUBTRACT from XREG".

**DECX MACR**

```
07390 STA TEMPAS
07400 LDA XREG1$+1
07410 SUB #1
07420 STA XREG1$+1
07430 STA XREG2$+1
07440 LDA XREG1$
07450 SBC #0
07460 STA XREG1$
07470 STA XREG2$
07480 ORA XREG1$+1
07490 STA TESTAS
07500 LDA TEMPA$
07510 TST TESTA$
07520 MEXIT
07530 ENDC
```

```
07540 IFEQ NARG
07550 STA TEMPAS
07560 LDA XREG1$+1
07570 SUB (#0)+1
07580 STA XREG1$+1
07590 STA XREG2$+1
07600 LDA XREG1$
07610 SBC #0
07620 STA XREG1$
07630 STA XREG2$
07640 ORA XREG1$+1
07650 STA TESTA$
07660 LDA TEMPA$
07670 TST TESTA$
07680 MEXIT
07690 ENDC
```
IFEQ NARG-2
    IFC '\0', '\0'
    STA TEMPA$
    LDA XREG1$+1
    SUB #\{1\}!.$FF
    STA XREG1$+1
    STA XREG2$+1
    LDA XREG1$
    SBC #\{1\}>8
    STA XREG1$
    STA XREG2$
    ORA XREG1$+1
    STA TESTA$
    LDA TEMPA$
    STA TESTA$
    STA XREG2$
    STA TENPA$
    MEXIT
    ENDC
    ENDC
    FAIL Macro syntax error detected!
    ENDM

07700 IFEQ NARG-2
07710 IFC '\0', '\0'
07720 STA TEMPA$
07730 LDA XREG1$+1
07740 SUB #\{1\}!.$FF
07750 STA XREG1$+1
07760 STA XREG2$+1
07770 LDA XREG1$
07780 SBC #\{1\}>8
07790 STA XREG1$
07800 STA XREG2$
07810 ORA XREG1$+1
07820 STA TESTA$
07830 LDA TEMPA$
07840 STA TESTA$
07850 MEXIT
07860 ENDC
07870 ENDC
07880 FAIL Macro syntax error detected!
07890 ENDM

07900 ******************************************************
07910 CPXR = compare XREG
07920 CPXR [#,] <address>
07930
07940
07950 * Examples:
07960 1. "CPXR #,BLOCKSZ" compares the value of symbol 'BLOCKSZ'
07970 with the XREG.
07980 2. "CPXR START" compares the contents of location
07990 'START' and 'START'+1 with the XREG.
08000
08010 * Register Usage:
08020 * CC = reflects XREG comparison (Z-bit only).
08030 * All other registers preserved.
08040
08050 CPXR MACR
08060 IFEQ NARG-1
08070 STA TEMPA$
08080 BSET 0,TESTA$
08090 LDA XREG1$+1
08100 CMP \{0\}+1
08110 BNE \.0
08120 LDA XREG1$
08130 CMP \{0\}
08140 BNE \.0
08150 CLR TESTA$
08160 \.0 LDA TEMPA$
08170 TST TESTA$
08180 MEXIT
08190 ENDC

439
08200  IFEQ  NARG-2
08210  IFC  \0\', \#'
08220  IFEQ  \1
08230  IFEQ  XREG1$!.$FF00
08240  TST  XREG1$+1
08250  BNE  \.0  Branch if LS half is .NE.
08260  TST  XREG1$
08270  \.0  EQU *
08280  MEXIT
08290  ENDC
08300  IFNE  XREG1$!.$FF00
08310  STA  TEMPAS$
08320  BSET  0.TESTAS$  Preset for .NE. condition!
08330  LDA  XREG1$+1
08340  BNE  \.0  Branch if MS half is .NE.
08350  LDA  XREG1$
08360  BNE  \.0  Branch if MS half is .NE.
08370  CLR  TESTAS$  Set for .EQ. condition!
08380  \.0  LDA  TEMPAS$
08390  TST  TESTAS$  Set proper Z-bit (.EQ. or .NE.)!
08400  MEXIT
08410  ENDC
08420  ENDC
08430  STA  TEMPAS$
08440  BSET  0.TESTAS$  Preset for .NE. condition!
08450  LDA  XREG1$+1
08460  IFNE  (\1)!.$FF
08470  CMP  #(\1)!.$FF
08480  ENDC
08490  BNE  \.0  Branch if LS half is .NE.
08500  LDA  XREG1$
08510  IFNE  (\1)!>8
08520  CMP  #(\1)!>8
08530  ENDC
08540  BNE  \.0  Branch if MS half is .NE.
08550  CLR  TESTAS$  Set for .EQ. condition!
08560  \.0  LDA  TEMPAS$
08570  TST  TESTAS$  Set proper Z-bit (.EQ. or .NE.)!
08580  MEXIT
08590  ENDC
08600  ENDC
08610  FAIL  Macro syntax error detected!
08620  ENDM
08630  440
08640 ****************************
08650 * LDYR = load YREG
08660 * LDYR  [#],<address>
08670 *
08680 * Examples:
08690 * 1. "LDYR #,START" puts the value of symbol 'START' into the
08700 *        YREG
08710 * 2. "LDYR START" puts the contents of location 'START' and
08720 *        'START'+1 into the YREG.
08730 *
08740 * Register Usage:
08750 * CC = reflects MS half.
08760 * All other registers preserved.
08770 *
08780 LDYR MACR
08790 IFEQ NARG-1
08800     STA  TEMPAS$  
08810     LDA  (\0)  
08820     STA  YREG1$  
08830     STA  YREG2$  
08840     LDA  (\0)+1  
08850     STA  YREG1$+1  
08860     STA  YREG2$+1  
08870 IFEQ  YREG1$!.FF00  
08880     LDA  TEMPAS$  
08890     TST  YREG1$  
08900   ENDC  
08910 IFNE  YREG1$!.FF00  
08920     STA  TESTA$  
08930     LDA  TEMPAS$  
08940     TST  TESTA$  
08950   ENDC  
08960  MEXIT  
08970 ENDC  
08980 IFEQ  NARG-2  
08990   IFC  '\0', '#'  
09000   IFEQ  YREG1$!.FF00  ! YREG in low memory?  
09010    IFEQ  \1  ! #0 value?  
09020     CLR  YREG1$+1  
09030     CLR  YREG2$+1  
09040     CLR  YREG1$  
09050     CLR  YREG2$  
09060  MEXIT  
09070 ENDC  
09080 IFNE  \1  ! not #0 value?  
09090     STA  TEMPAS$  
09100    IFEQ  (\1)!.$FF  
09110     CLR  YREG1$+1  
09120     CLR  YREG2$+1  
09130  ENDC  
09140 IFNE  (\1)!.$FF  
09150     LDA  #(\1)!.$FF  
09160     STA  YREG1$+1  
09170     STA  YREG2$+1  
09180  ENDC  
09190 IFEQ  (\1)!.>8  
09200     CLR  YREG1$  
09210     CLR  YREG2$  
09220  ENDC  
09230 IFNE  (\1)!>.>8  
09240     LDA  #\(\1)!>.>8  
09250     STA  YREG1$  
09260     STA  YREG2$  
09270  ENDC

441
LDA TEMPAS
TST YREG1$
MEXIT
ENDC
ENDC
IFNE YREG1$+.SFF00 ! YREG in high memory?
IFEQ \
STA TEMPAS
CLRA
STA YREG1$+1
STA YREG2$+1
STA YREG1$
STA YREG2$
CLR TESTA$
LDA TEMPAS
TST TESTA$
MEXIT
ENDC
IFNE \1 ! #0 value?
STA TEMPAS
IFEQ (\1)!+.SFF
CLRA
ENDC
IFNE (\1)!+.SFF
LDA #((\1)!+.SFF
ENDC
STA YREG1$+1
STA YREG2$+1
IFEQ (\1)!>8
CLRA
ENDC
IFNE (\1)!>8
LDA #((\1)!>8
ENDC
STA YREG1$
STA YREG2$
STA TESTA$
LDA TEMPAS
TST TESTA$
MEXIT
ENDC
ENDC
ENDC
ENDC
FAIL Macro syntax error detected!
ENDM
-----------------------------------------------------------------------------------
STYR = store YREG
STYR <address>

Examples:
1. "STYR START" stores the YREG into locations 'START' and 'START'+1.

Register Usage:
CC = reflects MS half.
All other registers preserved.

STYR MACR
STA TEMPAS
LDA YREG1$
STA (\0)
LDA YREG1$+1
STA (\0)+1
**INCYR** = increment YREG

### Examples:
1. **"INCYR"** adds one (1) to the YREG.
2. **"INCYR #,START"** adds the value of symbol 'START' to the YREG.
3. **"INCYR START"** adds the contents of location 'START' and 'START'+1 to the YREG.
4. **"INCYR ! comment"** adds one (1) to the YREG (comment present!).

### Register Usage:
- CC = reflects value incremented (Z-bit only).
- All other registers preserved.

### Notes:
1. Explicit comment character (') MUST be used when comment field is present.
2. Assumes YREG1$ = YREG2$.
3. When parameters are present, this macro becomes "ADD to YREG".

---

```
09930  IFEQ  YREG1$!, $FF00
09940   LDA  TEMPAS
09950   TST  YREG1$
09960  ENDC
09970  IFNE  YREG1$!, $FF00
09980   STA  TESTAS
09990   LDA  TEMPAS
10000  TST  TESTAS
10010  ENDC
10020  ENDM
10030
10040  *********************************************************************************************
10050  * INCYR = increment YREG
10060  * INCYR [[#]<value>]
10070  *
10080  * Examples:
10090  * 1. "INCYR" adds one (1) to the YREG.
10100  * 2. "INCYR #,START" adds the value of symbol 'START' to the YREG.
10110  * 3. "INCYR START" adds the contents of location 'START' and 'START'+1 to the YREG.
10120  * 4. "INCYR ! comment" adds one (1) to the YREG (comment present!).
10130  *
10140  * Register Usage:
10150  * CC = reflects value incremented (Z-bit only).
10160  * All other registers preserved.
10170  *
10180  * Notes:
10190  * 1. Explicit comment character (') MUST be used when comment field is present.
10200  * 2. Assumes YREG1$ = YREG2$.
10210  * 3. When parameters are present, this macro becomes "ADD to YREG".
10220  *
10230  *
10240  * INCYR MACR
10250 10260  IFEQ  NARG
10270 10280   IFEQ  YREG1$!, $FF00
10290   INC  YREG1$+1
10300   INC  YREG2$+1
10310   BNE  \.0
10320   INC  YREG1$
10330   INC  YREG2$
10340  \.0  EQU  *
10350  MEXIT
10360  ENDC
10370  IFNE  YREG1$!, $FF00
10380   STA  TEMPAS
10390   LDA  YREG1$+1
10400   ADD  #1
10410   STA  YREG1$+1
10420   STA  YREG2$+1
10430   LDA  YREG1$
10440   ADC  #0
10450   STA  YREG1$
10460   STA  YREG2$
10470   ORA  YREG1$+1
10480   STA  TESTAS
10490   LDA  TEMPAS
10500   TST  TESTAS
10510  MEXIT
10520  ENDC
10530  ENDC
```
10540 IFEQ NARG-1
10550   STA TEMPAS
10560   LDA YREG1$+1
10570   ADD (\0)+1
10580   STA YREG1$+1
10590   STA YREG2$+1
10600   LDA YREG1$
10610   ADC \0
10620   STA YREG1$
10630   STA YREG2$
10640   ORA YREG1$+1
10650   STA TESTA$
10660   LDA TEMPAS
10670   TST TESTA$
10680   MEXIT
10690 ENDC
10700 IFEQ NARG-2
10710 IFC '\0', '\'
10720   STA TEMPAS
10730   LDA YREG1$+1
10740   ADD #(1)!.FF
10750   STA YREG1$+1
10760   STA YREG2$+1
10770   LDA YREG1$
10780   ADC #(1)!>8
10790   STA YREG1$
10800   STA YREG2$
10810   ORA XREG1$+1
10820   STA TESTA$
10830   LDA TEMPAS
10840   TST TESTA$
10850   MEXIT
10860 ENDC
10870
10880   FAIL Macro syntax error detected!
10890 ENDM
10900
10910 ******************************************************************************
10920 * DECYR = decrement YREG
10930 * DECYR [[#,]<value>]
10940 *
10950 * Examples:
10960 * 1. "DECYR" subtracts one (1) from the YREG.
10970 * 2. "DECYR #,START" subtracts the value of symbol 'START' from
10980 * the YREG
10990 * 3. "DECYR START" subtracts the contents of location 'START'
11000 * and 'START'+1 from the YREG.
11010 * 4. "DECYR ! comment" subtracts one from the YREG (comment present!).
11020 *
11030 * Register Usage:
11040 * CC = reflects value decremented (Z-bit only).
11050 * All other registers preserved.
11060 *
11070 * Notes:
11080 * 1. Explicit comment character (!) MUST be used when comment field is
11090 * present!
11100 * 2. Assumes YREG1$ = YREG2$.
11110 * 3. When parameters are present, this macro becomes "SUBTRACT from YREG".
11120 *

444
11130 DECYR MACR
11140 IFEQ NARG
11150 STA TEMPA$
11160 LDA YREG1$+1
11170 SUB #1
11180 STA YREG1$+1
11190 STA YREG2$+1
11200 LDA YREG1$
11210 SBC #0
11220 STA YREG1$
11230 STA YREG2$
11240 ORA YREG1$+1
11250 STA TESTA$
11260 LDA TEMPA$
11270 TST TESTA$
11280 MEXIT
11290 ENDC
11300 IFEQ NARG-1
11310 STA TEMPA$
11320 LDA YREG1$+1
11330 SUB ($0)+1
11340 STA YREG1$+1
11350 STA YREG2$+1
11360 LDA YREG1$
11370 SBC \0
11380 STA YREG1$
11390 STA YREG2$
11400 ORA YREG1$+1
11410 STA TESTA$
11420 LDA TEMPA$
11430 TST TESTA$
11440 MEXIT
11450 ENDC
11460 IFEQ NARG-2
11470 IFC \0,'$'
11480 STA TEMPA$
11490 LDA YREG1$+1
11500 SUB #($1).$FF
11510 STA YREG1$+1
11520 STA YREG2$+1
11530 LDA YREG1$
11540 SBC #($1)>8
11550 STA YREG1$
11560 STA YREG2$
11570 ORA YREG1$+1
11580 STA TESTA$
11590 LDA TEMPA$
11600 TST TESTA$
11610 MEXIT
11620 ENDC
11630 ENDC
11640 FAIL Macro syntax error detected!
11650 ENDM
11660
445
** EXAMPLES: **

1. "CPYR #.BLOCKSZ" compares the value of symbol 'BLOCKSZ'
2. "CPYR START" compares the contents of location 'START' and 'START'+1 with the YREG.

** REGISTER USAGE:**
- CC = reflects YREG comparison (Z-bit only).
- All other registers preserved.

** MACRO DEFINITION:**

```
CPYR MACR

STA TEMPA$ Preset for .NE. condition!
BSET 0,TESTA$ Branch if MS half is .NE.
LDA YREG1$+1
CMP \0 Branch if MS half is .NE.
BNE \0 Branch if MS half is .NE.
CLR TESTA$ Set for .EQ. condition!
LDA TEMPA$ TST TESTA$ Set proper Z-bit (.EQ. or .NE.)!
TENPA$ Preset for .NE. condition!
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
CLR TESTA$ Set for .EQ. condition!
LDA TEMPA$ TST TESTA$ Set proper Z-bit (.EQ. or .NE.)!
TENPA$ Preset for .NE. condition!
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
CLR TESTA$ Set for .EQ. condition!
LDA TEMPA$ TST TESTA$ Set proper Z-bit (.EQ. or .NE.)!
TENPA$ Preset for .NE. condition!
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
CLR TESTA$ Set for .EQ. condition!
LDA TEMPA$ TST TESTA$ Set proper Z-bit (.EQ. or .NE.)!
TENPA$ Preset for .NE. condition!
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
CLR TESTA$ Set for .EQ. condition!
LDA TEMPA$ TST TESTA$ Set proper Z-bit (.EQ. or .NE.)!
TENPA$ Preset for .NE. condition!
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
CLR TESTA$ Set for .EQ. condition!
LDA TEMPA$ TST TESTA$ Set proper Z-bit (.EQ. or .NE.)!
TENPA$ Preset for .NE. condition!
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
CLR TESTA$ Set for .EQ. condition!
LDA TEMPA$ TST TESTA$ Set proper Z-bit (.EQ. or .NE.)!
TENPA$ Preset for .NE. condition!
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
BNE \0 Branch if LS half is .NE.
LDA YREG1$+1
CMP \0 Branch if LS half is .NE.
DEC.B = decrement byte

* DEC.B [[#,]<value>,]<address>

* where:

* <value> = value to decrement the contents of the <address>

* location by: immediate ("#," present) or absolute

* addressing ("#," not present). If only <address> is

* specified, a default immediate value of one is used.

* Examples:

* 1. "DEC.B START" subtracts one from the contents of

* 2. "DEC.B #,5,START" subtracts five from the contents of

* 3. "DEC.B CNT,START" subtracts the contents of location 'CNT'

* from the contents of location 'START'.

* Register Usage:

* CC = reflects value decremented (N and Z-bits).

* All other registers preserved.

* Notes:

* 1. <address> may be direct or extended!

* 2. This macro essentially performs a "SUB n" function.

DEC.B MACR

IFEQ NARG-1

IFEQ (\0)!.\FF00

DEC \0

MEXIT

STA TEMPA$

LDA \0

SUB \#1

STA \0

STA TESTA$

LDA TEMPA$

TST TESTA$

MEXIT

ENDC

ENDC

DEC

STA TEMPA$

LDA \0

SUB \#1

STA \0

IFEQ (\1)!.\FF00

LDA TEMPA$

TST \1

MEXIT

ENDC

IFEQ NARG-2

STA TEMPA$

LDA \1

SUB \0

STA \1

IFEQ (\1)!.\FF00

LDA TEMPA$

TST \1

MEXIT

ENDC
DEC.W = decrement word

**DEC.W** [[#,]<value>,]<address>

where:

* <value> = value to decrement the contents of the <address> and <address>+1 locations by; immediate ("#", present) or absolute addressing ("#", not present). If only <address> is specified, a default immediate value of one is used.

Examples:

1. "DEC.W START" subtracts one from the contents of locations 'START' and 'START'+1.
2. "DEC.W CNT,START" subtracts the contents of locations 'CNT' and 'CNT'+1 from the contents of locations 'START' and 'START'+1.
3. "DEC.W #,5,START" subtracts five from the contents of locations 'START' and 'START'+1.

Register Usage:

* CC = reflects value incremented (Z-bit only).
* All other registers preserved.

Notes:

1. <address> may be direct or extended!
2. This macro essentially performs a "SUB n" function.
DEC.W MACR
13490     IFEQ NARG-1
13500        STA TEMPA$
13510       LDA \(0)+1
13520       SUB #1
13530        STA \(0)+1
13540       LDA \0
13550      SBC #0
13560        STA \0
13570       ORA \(0)+1
13580        STA TESTA$
13590       LDA TEMPA$
13600      TST TESTA$
13610    MEXIT
13620  ENDC
13630     IFEQ NARG-2
13640        STA TEMPA$
13650       LDA \(1)+1
13660       SUB \(0)+1
13670        STA \(1)+1
13680       LDA \1
13690      SBC \0
13700        STA \1
13710       ORA \(1)+1
13720        STA TESTA$
13730       LDA TEMPA$
13740      TST TESTA$
13750    MEXIT
13760  ENDC
13770     IFEQ NARG-3
13780     IFC \0', '#'
13790        STA TEMPA$
13800       LDA \(2)+1
13810       SUB #\(1)!.FF
13820        STA \(2)+1
13830       LDA \2
13840      SBC #\(1)!>8
13850        STA \2
13860       ORA \(2)+1
13870        STA TESTA$
13880       LDA TEMPA$
13890      TST TESTA$
13900    MEXIT
13910  ENDC
13920  ENDC
13930     FAIL Macro syntax error detected!
13940    ENDM
13950
INC.B = increment byte

INC.B ([#],<value>,]<address>

where:

<value> = value to decrement the contents of the <address>

location by; immediate ("#," present) or absolute addressing ("#," not present). If only <address> is specified, a default immediate value of one is used.

Examples:

1. "INC.B START" adds one to the contents of location 'START'.
2. "INC.B #,5,START" adds five to the contents of location 'START'.
3. "INC.B CNT,START" adds the contents of location 'CNT' to the contents of location 'START'.

Register Usage:

CC= reflects value incremented (N and Z-bits).
All other registers preserved.

Notes:

1. <address> may be direct or extended!
2. This macro essentially performs an "ADD n" function.

INC.B MACR

IFEQ NARG-1

IFEQ (\0)!.$FF00
INC \0
MEXIT
ENDC

STA TEMPA$
LDA \0
ADD #1
STA \0
STA TESTA$
LDA TEMPA$
TST TESTA$
MEXIT
ENDC

IFEQ NARG-2

STA TEMPA$
LDA \1
ADD \0
STA \1
IFEQ (\1)!.$FF00
LDA TEMPA$
TST \1
MEXIT
ENDC

IFEQ (\1)!.$FF00
STA TESTA$
LDA TEMPA$
TST TESTA$
MEXIT
ENDC

ENDC
14540  IFEQ  NARG-3
14550  IFC  \0', '#'
14560  STA  TEMPAS$  
14570  LDA  \
14580  ADD  \
14590  STA  \
14600  IFEQ  (\2)!, $FF00
14610  LDA  TEMPAS$
14620  TST  \
14630  MEXIT
14640  ENDC
14650  IFNE  (\2)!, $FF00
14660  STA  TESTAS$
14670  LDA  TEMPAS$
14680  TST  TESTAS$
14690  MEXIT
14700  ENDC
14710  ENDC
14720  ENDC
14730  FAIL  Macro syntax error detected!
14740  ENDM
14750
14760  ************************************************************************************
14770  * INC.W = increment word
14780  * INC.W  [{\#,}]<value>,]<address>
14790  *
14800  * where:
14810  * <value>  = value to increment the contents of the <address> and
14820  * <address>+1 locations by: immediate ('\#,' present)
14830  * or absolute addressing ('\#,' not present). If only
14840  * <address> is specified, a default immediate value of
14850  * one is used.
14860  *
14870  * Examples:
14880  * 1. "INC.W START"  adds one to the contents of locations
14890  * 2. "INC.W CNT,START"  adds the value of 'CNT' to the contents
14900  * 3. "INC.W #,5,START"  adds five to the contents of locations
14910  * 4. "INC.W,#,START"  adds one to the contents of locations
14920  * 5. "INC.W,#,START"  adds one to the contents of locations
14930  *
14940  *
14950  * Register Usage:
14960  * CC  = reflects value incremented (Z-bit only).
14970  * All other registers preserved.
14980  *
14990  * Notes:
15000  * 1. <address> may be direct or extended!
15010  * 2. This macro essentially performs an "ADD n" function.
15020  *
15030  INC.W  MACR
15040  IFEQ  NARG-1
15050  IFEQ  (\0)!, $FF00
15060  INC  (\0)+1
15070  BNE  \
15080  INC  \
15090  \.0  EQU  *
15100  MEXIT
15110  ENDC
15120     IFNE  (\0)!.$FF00
15130         STA    TEMPS$  
15140         LDA    (\0)+1  
15150         ADD    #1  
15160         STA    (\0)+1  
15170         LDA    \0  
15180         ADC    #0  
15190         STA    \0  
15200         ORA    (\0)+1  
15210         STA    TESTA$  
15220         LDA    TEMPS$  
15230         TST    TESTA$  
15240         MEXIT  
15250         ENDC  
15260         ENDC  
15270     IFEQ    NARG-2  
15280         STA    TEMPS$  
15290         LDA    (\1)+1  
15300         ADD    (\0)+1  
15310         STA    (\1)+1  
15320         LDA    \1  
15330         ADC    \1  
15340         STA    \1  
15350         ORA    (\1)+1  
15360         STA    TESTA$  
15370         LDA    TEMPS$  
15380         TST    TESTA$  
15390         MEXIT  
15400         ENDC  
15410     IFEQ    NARG-3  
15420     IFC     \0,"#$  
15430         STA    TEMPS$  
15440         LDA    (\2)+1  
15450         ADD    #(\1)!.$FF  
15460         STA    (\2)+1  
15470         LDA    \2  
15480         ADC    #(\1)!>8  
15490         STA    \2  
15500         ORA    (\2)+1  
15510         STA    TESTA$  
15520         LDA    TEMPS$  
15530         TST    TESTA$  
15540         MEXIT  
15550         ENDC  
15560         ENDC  
15570     FAIL   Macro syntax error detected!  
15580     ENDM
**MOV.B = move byte**

**where:**

- `<byte>` = byte value to move to the `<address>` location, using immediate ("#," present) or absolute addressing ("lI," not present).

**Examples:**

1. "MOV.B CNT, TMP" puts the contents of location 'CNT'
2. "MOV.B #, START" puts 5 into location 'START'.

**Register Usage:**

- puts the contents of location 'CNT'
- puts 5 into location 'START'.
- CC = reflects value moved.
- All other registers preserved.
16230 IFNE \(\{2\}!:.FF00
16240 STA TESTA$
16250 LDA TEMPAS
16260 TST TESTA$
16270 MEXIT
16280 ENDC
16290 ENDC
16300 ENDC
16310 FAIL Macro syntax error detected!
16320 ENDM
16330
********************************************************************************
16350 * MOV.W = move word
16360 * MOV.W [#]<word>,<address>
16370 *
16380 * where:
16390 * <word> = word (16-bit) value to move to the <address> and
16400 * <address>+1 locations, using immediate ("#," present)
16410 * or absolute addressing ("#," not present).
16420 *
16430 * Examples:
16440 1. "MOV.W #,START" puts $0005 into location 'START' and
16450 * 'START'+1.
16460 2. "MOV.W #,CNT,TMP" puts the value of symbol 'CNT' into
16470 * locations 'TMP' and 'TMP'+1.
16480 3. "MOV.W CNT,TMP" copies the contents of location 'CNT'
16490 * and 'CNT'+1 into locations 'TMP' and
16500 * 'TMP'+1.
16510 *
16520 * Register Usage:
16530 * CC = reflects MS half of value moved.
16540 * All other registers preserved.
16550 *
16560 MOV.W MACR
16570 IFEQ NARG-2
16580 STA TEMPAS
16590 LDA \(\{0\}+1
16600 STA \(\{1\}+1
16610 LDA \0
16620 STA \1
16630 IFEQ \(\{1\}!:.FF00
16640 LDA TEMPA$
16650 TST \1
16660 MEXIT
16670 ENDC
16680 IFNE \(\{1\}!:.FF00
16690 IFEQ \(\{0\}!:.FF00
16700 LDA TEMPA$
16710 TST \0
16720 MEXIT
16730 ENDC
16740 IFNE \(\{0\}!:.FF00
16750 STA TESTA$
16760 LDA TEMPAS
16770 TST TESTA$
16780 MEXIT
16790 ENDC
16800 ENDC
16810 ENDC

454
16820 IFEQ NARG-3
16830 IFC '0', #'
16840 IFEQ \((\2)+1\) !$.FF00
16850 IFEQ \1
16860 CLR \2
16870 CLR \2+1
16880 MEXIT
16890 ENDC
16900 ENDC
16910 STA TEMP$A$
16920 IFEQ \((\1)!$.00FF
16930 CLRA
16940 ENDC
16950 IFNE \((\1)!$.00FF
16960 LDA #\((\1)!$.00FF
16970 ENDC
16980 STA \((\2)+1
16990 IFEQ \((\1)!>8
17000 IFNE \((\1)!$.00FF
17010 CLRA
17020 ENDC
17030 ENDC
17040 IFNE \((\1)!>8
17050 LDA #\((\1)!>8
17060 ENDC
17070 STA \2
17080 IFEQ \((\2)!$.FF00
17090 LDA TEMPA$
17100 TST \2
17110 MEXIT
17120 ENDC
17130 IFEQ \((\2)!$.FF00
17140 STA TESTA$
17150 LDA TEMPA$
17160 TST TESTA$
17170 MEXIT
17180 ENDC
17190 ENDC
17200 ENDC
17210 FAIL Macro syntax error detected!
17220 ENDM
17230
MOVE = move block of memory

MOVE [##],<source>[##],<destination>[##],[###]<length>

where:

<source> is the address of the source memory block.

<destination> is the address of the destination block.

[length] is the length of the block to move, in bytes.

Maximum of 65,536 bytes can be moved.

# is optional character to denote immediate addressing for the next parameter

Examples:

1. MOVE #,ROM,#,RAM,#,CNT moves the block of memory starting at location 'ROM' for 'CNT' bytes, to location 'RAM'.

2. MOVE #,ABC,#,XYZ,,CNT moves the block of memory starting at location 'ABC' for the number of bytes in locations 'CNT' and 'CNT'+1, to location 'XYZ'.

Register Usage:

CC = unknown.

All other registers preserved.

Subr. used:

LDAXREG, STAXREG

Macros used:

None, because this macro was written to be as efficient as possible.

Notes:

1. If all immediate addressing operands (#) and the move count is <= 256, then a special 'short form' is generated which DOES NOT contain any subroutine calls!

2. Depending on the exact parameters passed, not all registers, subroutines and/or macros may be used.

3. This macro takes advantage of the fact that there are in fact two XREGs, one for LOAD (XREG1$) and one for STORE (XREG2$).

4. The INCR macro cannot be used here, because it assumes that XREG1$ = XREG2$.

MOVE MACR

IFN...
17890 STA TEMPAS$
17900 STX TEMPSX$
17910 LDA XREG1$
17920 STA TEMPXR$
17930 LDA XREG1$+1
17940 STA TEMPXR$+1
17950 IFC '"0', '"'
17960 LDA #("1")! .SFF
17970 STA XREG1$+1
17980 LDA #("1")! >8
17990 STA XREG1$
18000 ENDC
18010 IFNC '"0', '"'
18020 LDA ("1")+1
18030 STA XREG1$+1
18040 LDA ("1")
18050 STA XREG1$
18060 ENDC
18070 IFC '"2', '"'
18080 LDA #("3")! .SFF
18090 STA XREG2$+1
18100 LDA #("3")! >8
18110 STA XREG2$
18120 ENDC
18130 IFNC '"2', '"'
18140 LDA ("3")+1
18150 STA XREG2$+1
18160 LDA ("3")
18170 STA XREG2$
18180 ENDC
18190
18200 IFC '"4', '"'
18210 IFEQ 5-256
18220 LDX #("5")
18230 .0 JSR LDAXREG
18240 JSR STAXREG
18250 IFEQ XREG1$!.SFF00
18260 INC XREG1$+1
18270 BNE \.1
18280 INC XREG1$
18290 \.1 INC XREG2$+1
18300 BNE \.2
18310 INC XREG2$
18320 \.2 EQU *
18330 ENDC
18340 IFNE XREG1$!.SFF00
18350 LDA XREG1$+1
18360 ADD #1
18370 STA XREG1$+1
18380 LDA XREG1$
18390 ADC #0
18400 STA XREG1$
18410 LDA XREG2$+1
18420 ADD #1
18430 STA XREG2$+1
18440 LDA XREG2$
18450 ADC #0
18460 STA XREG2$
18470 ENDC
18480 DEX
18490 BNE \.0
18500 LDA TEMPXR$
18510 STA XREG1$
18520 STA XREG2$
18530 LDA TEMPXR$+1

457
18540  STA  XREG1$+1
18550  STA  XREG2$+1
18560  LDA  TEMPA$
18570  LDX  TEMPX$
18580  MEXIT
18590  ENDC
18600  *
18610  IFGT  \5-256  !  no: 16-bit size= use 'length'
18620  LDA  #\(5)!.$00FF
18630  STA  LENGTH$+1
18640  LDA  #\(5)!>8
18650  STA  LENGTH$
18660  \0  JSR  LDAXREG
18670  JSR  STAXREG
18680  IFEQ  XREG1$!.$FF00
18690  INC  XREG1$+1
18700  BNE  \.1
18710  INC  XREG1$
18720  \.1  INC  XREG2$+1
18730  BNE  \.2
18740  INC  XREG2$
18750  \.2  EQU  *
18760  ENDC
18770  IFNE  XREG1$!.$FF00
18780  LDA  XREG1$+1
18790  ADD  #1
18800  STA  XREG1$+1
18810  LDA  XREG1$
18820  ADC  #0
18830  STA  XREG1$
18840  LDA  XREG2$+1
18850  ADD  #1
18860  STA  XREG2$+1
18870  LDA  XREG2$
18880  ADC  #0
18890  STA  XREG2$
18900  ENDC
18910  LDA  LENGTH$+1
18920  SUB  #1
18930  STA  LENGTH$+1
18940  LDA  LENGTH$
18950  SBC  #0
18960  STA  LENGTH$
18970  ORA  LENGTH$+1
18980  BNE  \.0
18990  LDA  TEMPRX$
19000  STA  XREG1$
19010  STA  XREG2$
19020  LDA  TEMPRX$+1
19030  STA  XREG1$+1
19040  STA  XREG2$+1
19050  LDA  TEMPA$
19060  LDX  TEMPX$
19070  MEXIT
19080  ENDC
19090  ENDC
19100

458
19110 IFNC \4', '#'
          ! nonimmediate type length
19120   LDA (\5)!.$00FF
19130   STA LENGTH$+1
19140   LDA (\5)!>$8
19150   STA LENGTH$
19160 \0  JSR LDAXREG
19170   JSR STAXREG
19180  IFEQ XREG1$!.$FF00
19190   INC XREG1$+1
19200   BNE \1
19210   INC XREG1$
19220 \1  INC XREG2$+1
19230   BNE \2
19240   INC XREG2$
19250 \2  EQU *
19260  ENDC
19270 IFNE XREG1$!.$FF00
19280   LDA XREG1$+1
19290   ADD #1
19300   STA XREG1$+1
19310   LDA XREG1$
19320   ADC #0
19330   STA XREG1$
19340   LDA XREG2$+1
19350   ADD #1
19360   STA XREG2$+1
19370   LDA XREG2$
19380   ADC #0
19390   STA XREG2$
19400  ENDC
19410   LDA LENGTH$+1
19420   SUB #1
19430   STA LENGTH$+1
19440   LDA LENGTH$
19450   SBC #0
19460   STA LENGTH$
19470   ORA LENGTH$+1
19480 \0  BNE \0
19490   LDA TEMPXR$
19500   STA XREG1$
19510   STA XREG2$
19520   LDA TEMPXR$+1
19530   STA XREG1$+1
19540   STA XREG2$+1
19550   LDA TEMP$
19560   LDX TEMP$
19570  MEXIT
19580  ENDC
19590   FAIL Macro syntax error detected!
19600   ENDM
19610
19620
**Start of RAM based subroutines!**

The following RAM subroutines MUST BE INITIALIZED from ROM upon startup (from 'RAMSBR$' for 'RAMSZ$' number of bytes). If changes are to be made to the RAM subroutines, make them here. Then copy "\texttt{.RAMSBR,II,"} into your ROM area and insert a \texttt{.} in front of all the labels (leading \texttt{.} will be used to denote ROM). This has already been done for you in the \texttt{RAMSBR.INI} file. Just include this file into your ROM data area and add the following line in your \texttt{RESET} routine to initialize the RAM subroutines from the ROM.

\texttt{MOVE II, .RAMSBR,II,RAMSBR,II,}$

It is more efficient if the RAM subroutines are placed in DIRECT addressing memory, i.e., \texttt{$0000-$00FF}, but it is not required.

\texttt{************************************************.************************************}

--- start of RAM subroutines -----------------------------------------------

** Register Usage:**
\texttt{CC} reflects value loaded.
All other registers preserved.

*NOTE:*
1. Instruction modified code here must be located in RAM!

\texttt{LDAXREG = load A via XREG subr.}
\texttt{XREG1$ EQU *-2 Pseudo XREG #1}
\texttt{RTS}

**Register Usage:**
\texttt{CC} reflects value stored.
All other registers preserved.

*NOTE:*
1. Instruction modified code here must be located in RAM!

\texttt{STAXREG = store A via XREG subr.}
\texttt{XREG2$ EQU *-2 Pseudo XREG #2}
\texttt{RTS}

** Register Usage:**
\texttt{CC} reflects value loaded.
All other registers preserved.

*NOTE:*
1. Instruction modified code here must be located in RAM!

\texttt{LDAYREG = load A via YREG subr.}
\texttt{YREG1$ EQU *-2 Pseudo YREG #1}
\texttt{RTS}

**Register Usage:**
\texttt{CC} reflects value stored.
All other registers preserved.

*NOTE:*
1. Instruction modified code here must be located in RAM!

\texttt{LDAYREG = load A via YREG subr.}
\texttt{YREG1$ EQU *-2 Pseudo YREG #1}
\texttt{RTS}
**STAYREG**

* Register Usage:*
  - CC = reflects value stored.
  - All other registers preserved.

* NOTE:*
  - 1. Instruction modified code here must be located in RAM!

```assembly
20250 ; --------
20260 * STAYREG = store A via YREG subr.
20270 *
20280 * Register Usage:
20290 * CC = reflects value stored.
20300 * All other registers preserved.
20310 *
20320 * NOTE:
20330 * 1. Instruction modified code here must be located in RAM!
20340 *
20350 STAYREG EQU *
20360 STA 0-0+$FFFF
20370 YREG$ EQU -*-2 Pseudo YREG #2
20380 RTS
20390 /* end of RAM subroutines ----------------------------------------------- */
20400
20410 RAMSZ$ EQU -*-RAMSBR$ Size of ram subroutines (in bytes).
20420
20430 ORG LO$MEM
20440 /* NOTE: TENPA$ and TESTA$ must always be in low memory $0000-00FF.
20450 TEMP$ RMB 1 Temporary storage for A accumulator.
20460 TEMPX$ RMB 1 Temporary storage for X register.
20470 TEMPXR$ RMB 2 Temporary storage for XREG register.
20480 TESTA$ RMB 1 Temporary operand storage for setting CC bits.
20490 LENGTH$ RMB *2 Temporary operand length.
20500
20510 ; --------
```

---

461
Module Name: RAMSBR - RAM Subroutine Initialization

Description:
This file contains the initialization code for the RAM subroutine area needed to support the MACROS05.MAC file. It MUST be placed in the ROM data area and then copied to RAM for proper operation.

Consult the MACROS05.MAC file for more details.

Notes:
1. Motorola reserves the right to make changes to this file. Although this file has been carefully reviewed and is believed to be reliable, Motorola does not assume any liability arising out of its use. This code may be freely used and/or modified at no cost or obligation by the user.
2. The latest version of this file is maintained on the Motorola FREEWARE Bulletin Board, 512/891-FREE (512/891-3733). It operates continuously (except for maintenance) at 1200-2400 baud, 8 bits, no parity. Sample test files for PASM05 are also included. Download the archive file, MACROS05.ARC, to get all the files.

Listing 2 — RAMSBR.INI File

00010 **                            **************************************************
00020 * ramsbr.ini 1.0               **************************************************
00030 * Module Name: RAMSBR - RAM Subroutine Initialization
00040 *
00050 * Description: This file contains the initialization code for the RAM subroutine area needed to support the MACROS05.MAC file. It MUST be placed in the ROM data area and then copied to RAM for proper operation.
00060 * Consult the MACROS05.MAC file for more details.
00070 *
00080 * Notes:
00090 * 1. Motorola reserves the right to make changes to this file. Although this file has been carefully reviewed and is believed to be reliable, Motorola does not assume any liability arising out of its use. This code may be freely used and/or modified at no cost or obligation by the user.
00100 * 2. The latest version of this file is maintained on the Motorola FREEWARE Bulletin Board, 512/891-FREE (512/891-3733). It operates continuously (except for maintenance) at 1200-2400 baud, 8 bits, no parity. Sample test files for PASM05 are also included. Download the archive file, MACROS05.ARC, to get all the files.
00110 *
00120 * REVISION HISTORY (add new changes to top):
00130 *
00140 * 05/16/90 P.S. Gilmour * 1. Original entry generated from MACROS05.MAC version 1.0.
00150 *
00160 *
00170 *
00180 *
00190 *
00200 *
00210 *
00220 *
00230 *
00240 *
00250 *
00260 *
00270 *
00280 *
00290 *
00300 *
00310 * 05/16/90 P.S. Gilmour
00320 * 1. Original entry generated from MACROS05.MAC version 1.0.
00330 ****************************************************
00340 *
00350 .RAMSBR$ EQU * Start of RAM based subroutines!
00360 *****************************************************
00370 ** The following RAM subroutines MUST BE INITIALIZED from ROM upon **
00380 ** startup (from 'RAMSBR$' for 'RAMSZ$' number of bytes). If changes **
00390 ** are to be made to the RAM subroutines, make them in the MACROS05.MAC **
00400 ** file and then copy the source here (ROM area) and insert a '.' in **
00410 ** front of all the labels (leading '.' will be used to denote ROM). **
00420 *****************************************************
00430 *
00440 ** start of RAM subroutines ----------------------------------------
00450 *****************************************************
00460 * LDAXREG = load A via XREG subr.
00470 *
00480 * Register Usage:
00490 * CC = reflects value loaded.
00500 * All other registers preserved.
00510 *
00520 * NOTE:
00530 * 1. Instruction modified code here must be located in RAM!
00540 *
00550 .LDAXREG EQU *
00560 LDA 0-0+$FFFF
00570 .XREG1$ EQU **-2 Pseudo XREG #1
00580 RTS
00590 

462
00600  ************************************************************************************
00610 * STA$X = store A via XREG subr.
00620 *
00630 * Register Usage:
00640 * CC = reflects value stored.
00650 * All other registers preserved.
00660 *
00670 * NOTE:
00680 * 1. Instruction modified code here must be located in RAM!
00690 *
00700 .STA$X EQU *
00710 STA 0-0+$FFFF
00720 .XREG2$ EQU *-2  Pseudo XREG #2
00730 RTS
00740
00750  ************************************************************************************
00760 * LDAYREG = load A via YREG subr.
00770 *
00780 * Register Usage:
00790 * CC = reflects value loaded.
00800 * All other registers preserved.
00810 *
00820 * NOTE:
00830 * 1. Instruction modified code here must be located in RAM!
00840 *
00850 .LDAYREG EQU *
00860 LDA 0-0+$FFFF
00870 .YREG1$ EQU *-2  Pseudo YREG #1
00880 RTS
00890
00900  ************************************************************************************
00910 * STA$Y = store A via YREG subr.
00920 *
00930 * Register Usage:
00940 * CC = reflects value stored.
00950 * All other registers preserved.
00960 *
00970 * NOTE:
00980 * 1. Instruction modified code here must be located in RAM!
00990 *
01000 .STA$Y EQU *
01010 STA 0-0+$FFFF
01020 .YREG2$ EQU *-2  Pseudo YREG #2
01030 RTS
01040 *-- end of RAM subroutines -------------------------------------------------------- *
01050
01060 .RAMSZ$ EQU *-.RAMSBR$ .Size of ram subroutines (in bytes).
01070 PNE RAMSZ$-.RAMSZ$  Size mismatch between RAM/ROM subroutine areas!
01080 ENDC
Selecting the Right Microcontroller Unit

INTRODUCTION

Selecting the proper microcontroller unit (MCU) for your application is one of the critical decisions which control the success or failure of your project. There are numerous criteria to consider when choosing an MCU and this Application Note will enumerate most of them and presents an outline of the thought process guiding this decision. The reader must attach their own grading scale to the selection criteria presented and then evaluate the total to make the correct decision.

PURPOSE

The main goal is to select the least expensive MCU that minimizes the overall cost of the system while still fulfilling the system specification, i.e., performance, reliability, environmental, etc. The overall cost of the system includes everything, such as Engineering Research and Development (R&D), manufacturing (parts and labor), warranty repairs, updates, field service, upward compatibility, ease of use, etc.

SELECTION PROCESS

To start the selection process, the designer must first ask the question, "What does the MCU need to do in my system?" The answer to this one simple question dictates the required MCU features for the system and thus is the controlling agent in the selection process.

The second step is to conduct a search for MCUs which meet all of the system requirements. This usually involves searching the literature, primarily data books, data sheets, and technical trade journals, but also includes peer consultations. These days, recent trade journals seem to contain the most up to date information for the newer MCUs. If the fit is good enough, a single-chip MCU solution has been found, otherwise a second search must be conducted to find an MCU which best fits the requirements with a minimum of extra circuitry, including considerations of cost and board space. Obviously, a single-chip solution is preferred for cost as well as reliability reasons. Of course, if there is a company policy dictating which MCU manufacturer to use, this will narrow your search considerably.

The last step has several parts, all of which attempt to reduce the list of acceptable MCUs to a single choice. These parts include pricing, availability, development tools, manufacturer support, stability, and sole-sourcing. The whole process may need to be iterated several times to arrive at the optimum decision.

SELECTION CRITERIA

The general outline of the main criteria in selecting a microcontroller is listed below, in the order of importance. Each criterion is explained in greater detail later on.

1. Suitability for the application system, i.e., can it be done with a single-chip MCU or at most a few additional chips?
   A. Does it have the required number of I/O pins/ports, i.e., too few = can't do the job and too many = excessive cost?
   B. Does it have all the other required peripherals, such as serial I/O, RAM, ROM, A/D, D/A, etc.?
   C. Does it have other peripherals that are not needed?
   D. Does the CPU core have the correct throughput, i.e., computing power, to handle the system requirements over the life of the system for the chosen implementation language? Too much is wasteful and too little will never work.
   E. Is the MCU affordable, i.e., does the project budget allocate enough funds to permit using this MCU? A budgeting quotation from the manufacturer is usually required to answer this question. If the MCU is not affordable for the project, all the other questions become irrelevant and you must start looking for another MCU.

2. Availability?
   A. Is the device available in sufficient quantities?
   B. Is the device in production today?
   C. What about the future?

3. Development support available?
   A. Assemblers.
   B. Compilers.
   C. Debugging tools.
      1. Evaluation Module (EVM).
      2. In-circuit emulators.
      3. Logic analyzer pods
      4. Debug monitors
      5. Source level debug monitors.
   D. On-line bulletin board service (BBS).
      1. Real time executives.
      2. Application examples.
      4. Utility software, including "free" assemblers.
      5. Sample source code.
E. Applications support.
1. Specific group who does nothing but applications support?
2. Application engineers, technicians, or marketers?
3. How knowledgeable are the support personnel? Are they truly interested in helping you with your problem?
4. Telephone and/or FAX support?
4. Manufacturer's history, i.e., "track record."
   A. Demonstrated competence in design.
   B. Reliability of silicon, i.e., manufacturing excellence.
   C. On-time delivery performance.
   D. Years in business.
   E. Financial report.

SYSTEM REQUIREMENTS

Applying system analysis to the current project will determine the MCU requirements for the system. What peripheral devices are required? Is the application to be bit manipulating or number crunching? Once data is received, how much manipulation is required? Is the system to be driven by interrupt, polled, or human-responses? How many devices/bits (I/O pins) need to be controlled? Among the many possible types of I/O devices to be controlled/monitored are RS-232C terminals, switches, relays, keypads, sensors (temperature, pressure, light, voltage, etc.), audible alarms, visual indicators (LCD displays, LEDs), analog to digital (A/D), and digital to analog (D/A). Is a single or multiple voltage power supply required for the system? What is the power supply tolerance? Is the device characterized for operation at your system supply voltage? Are the voltages to be held to a small fixed percent variation or are they to operate over a wider range? What is the operating current? Is the product to be ac or battery operated? If battery operated, should rechargeables be used, and if so, what is the operational time required before recharging and the required time for recharging?

Are there size and weight restrictions or aesthetic considerations such as shape and/or color? Is there anything special about the operating environment, such as military specification, temperature, humidity, atmosphere (explosive, corrosive, particulates, etc.), pressure/altitude? Is the application to be disk-based or ROM-based? Is it a real time application, and if so, are you going to build or purchase a real time kernel program or maybe a public domain version will suffice? Does your schedule contain enough time and personnel to develop your own? What about royalty payments and bug support? Much more investigation is required for real time applications in order to evaluate their special requirements.

GENERAL MCU ATTRIBUTES

MCUs can generally be classified into 8-bit, 16-bit, and 32-bit groups based upon the size of their arithmetic and index registers(s), although some designers argue that bus access size determines the 8/16/32-bit architecture. Is a lower-cost 8-bit MCU able to handle the requirements of the system, or is a higher-cost 16-bit or 32-bit MCU required? Can 8-bit software simulation of features found on the 16-bit or 32-bit MCUs permit using the lower-cost 8-bit MCU by sacrificing some code size and speed? For example, can an 8-bit MCU be used with software macros to implement 16-bit accumulator and indexing operations? The choice of implementation language (high level vs assembler) can greatly affect system throughput, which can then dictate the choice of 8/16/32-bit architectures, but system cost restraints may override this.

Clock speed, or more accurately, bus speed, determines how much processing can be accomplished in a given amount of time by the MCU. Some MCUs have a narrow clock speed range, whereas others can operate down to zero. Sometimes a specific clock frequency is chosen in order to generate another clock required in the system, e.g., for serial baud rates. In general, computational power, power consumption and system cost increase with higher clock frequencies. System costs increase with frequency because not only does the MCU cost more, but so do all the support chips required, such as RAMs, ROMs, PLDs, and bus drivers.

Consider also the processing technology of the MPU; N-channel metal-oxide semiconductor (NMOS) vs high-density complementary metal-oxide semiconductor (HCMOS). In HCMOS, signals drive from rail-to-rail, unlike earlier NMOS processors. Since these criteria can significantly affect noise issues in system design, HCMOS processors are usually preferred. Also, HCMOS uses less power and thus generates less heat. The design geometries in HCMOS are smaller which permit denser designs for a given size, and thus allow higher bus speeds. The denser designs also allow lower cost, for more units can be processed on the same sized silicon wafer. For these reasons, most MCUs today are produced using HCMOS technology.

MCU RESOURCES

By definition, all MCUs have on-chip resources to achieve a higher level of integration and reliability at a lower cost. An on-chip resource is a block of circuitry built into the MCU which performs some useful function under control of the MCU. Built-in resources increase reliability because they do not require any external circuitry to be working for the resource to function. They are pre-tested by the manufacturer and conserve board space by integrating the circuitry into the MCU. Some of the more popular on-chip resources are memory devices, timers, system clock/oscillator, and I/O. Memory devices include read/write memory (RAM), read-only memory (ROM), erasable programmable ROM (EPROM), electrically erasable programmable ROM (EEPROM) and electrically erasable memory (EEM). The term EEM actually refers to an engineering development version of an MCU where EEPROM is substituted for the ROM in order to reduce development time. Timers include both real time clocks and periodic interrupt timers. Be sure to consider the range and resolution of the timer as well as any subfunctions, such as timer compare and/or input capture lines. I/O includes serial communication ports, parallel ports (I/O lines), analog-to-digital (A/D) converters, digital-to-analog (D/A) converters, liquid crystal display drivers (LCD), and vacuum fluorescent display drivers (VFD).

Other less common built-in resources are internal/external bus capability, computer operating properly (COP) watchdog system, clock operating properly detection, selectable memory configurations, and system integration module (SIM). The SIM replaces the external "glue" logic usually required to interface to external devices via chip select pins.

On most MCUs with on-chip resources, a configuration register block is included to control these resources. Sometimes the configuration register block itself can be set up to appear at a different location in the memory map. Sometimes a user...
and/or factory test register is present, which indicates concern for quality by the manufacturer. With configuration registers also comes the possibility of errant code altering the desired configuration, so check for “lock-out” mechanisms, i.e., before a register can be changed, a bit in another register must first be altered in a certain sequence. Although configuration registers can at first be very confusing and intimidating because of their complexity, they are extremely valuable because of the flexibility they offer at a low cost so that a single MCU can serve many applications.

**MCU INSTRUCTION SET**

The instruction set and registers of each MCU should be carefully considered, as they play critical roles in the capability of the system. Have software engineers study the indexed addressing modes versus the anticipated needs of your system. Are there any specialty instructions available which could be used in your system, such as multiply, divide, and table lookup/interpolate? Are there any low power modes for battery conservation, such as stop, low power stop, and/or wait? Are there any bit manipulation instructions (bit set, bit clear, bit test, bit change, branch on bit set, branch on bit clear) to allow easier implementation of controller applications? How about bit field instructions?

Be dubious of fancy instructions which seem to do a lot in one instruction. The real measure of performance is how many clock cycles it takes to accomplish the task at hand, not how many instructions were executed. A fair comparison is to code the same routine and compare the total number of clock cycles executed and bytes used. Are there any unimplemented instructions in the opcode map and what happens if they are accidentally executed? Does the system handle this gracefully with an exception handler or does the system crash?

**MCU INTERRUPTS**

Examining the interrupt structure is a necessity when constructing a real time system. How many interrupt lines or levels are there versus how many does your system require? Is there an interrupt level mask? Once an interrupt level is acknowledged, are there individual vectors to the interrupt handler routines or must each possible interrupt source be polled to determine the source of the interrupt? In speed critical applications, such as controlling a printer, the interrupt response time, i.e., the time from the start of the interrupt (worst case phasing relative to the MCU clock) until the first instruction in the appropriate interrupt handler is executed, can be the selection criterion in determining the right MCU.

**COMPANY ATTRIBUTES**

Examine the assets of your own company with a little truthful introspection. Does your company have a significant investment in knowledge/training of existing personnel with a particular MCU manufacturer and in the development tools for those MCUs? Does your company own enough development tools or will you have to buy or rent more? If a new MCU is under consideration, are there development tools available, such as high level language compilers, assembler/linkers, evaluation modules, and debuggers/emulators? Are your present development tools easily expandable for new MCUs? Will additional personnel have to be hired and trained for this project? Can you hire an expert to train/lead the rest of your team? Does your budget permit hiring additional permanent staff and/or contractors? Is your company satisfied with your current MCU manufacturer’s product line and services?

**SUPPLIER ATTRIBUTES**

The third step is pruning the list of technically acceptable MCUs by examining the MCU manufacturer and supplier, i.e., the companies with which you plan to enter into a long-term relationship for mutual benefit. A supplier can either be the MCU manufacturer itself, or it can be a full-service dealer who is the authorized representative for several manufacturers. A supplier with a broader range of products and a reputation for quality, reliability, service, and on-time deliverability at a fair price can best serve your needs. Additionally, the more products you purchase from one supplier, the more leverage you obtain for pricing, service, and support. Always keep in mind that although your dollar volume may seem high to you, it is always a relative amount to the total business of the supplier. Suppliers who can furnish not only MCUs, but memories (RAM and ROM), discrete devices (transistors, diodes, etc.), standard digital logic devices (7400, 74HC00, etc.), specialty chips, customer specific devices (CISC), application specific devices (ASIC), and programmable logic devices (PLDs), will be better suited to serve your growing needs. Has the manufacturer and/or supplier won any awards for quality, reliability, service, and/or deliverability? Be suspicious of self-bestowed awards.

**MANUFACTURER ATTRIBUTES**

Other criteria to consider in selecting the MCU manufacturer/supplier are stability, sole-supplier status, literature, and support. Stability can best be ascertained by considering the number of years in business and obtaining a Dunn & Bradstreet rating plus copies of past Annual and Quarterly Financial Reports. Your company’s Purchasing and Credit Departments can greatly assist you in these areas. Listing on a major stock exchange is another sign of stability. A local stock broker can assist you in obtaining up-to-date information for those manufacturers listed on stock exchanges, or you can visit your local library to check the Periodical Guide for pertinent information. The Wall Street Journal is another excellent source of up-to-date financial news. Sole-supplier status is unfortunately usually the norm, as most MCU manufacturers do not often cross license their products to other manufacturers. If the manufacturer has a good track record for supply, delivery, and pricing, sole-supplier status should not be a problem.

**MANUFACTURER SUPPORT**

Direct manufacturer support includes Marketing/Sales, Field Application Engineers (FAEs), and Application Engineering. Are the FAEs near your site? When telephoning for support, can you reach the support person directly or do you play “telephone tag”? Are calls returned promptly? Is there a toll-free 800 number? Is there a FAX number? How many phone lines are available? Are the phone lines always busy? Do they have an individual Voice Mail answering system or does a secretary in another office take “While You Were Out” messages which must be physically relayed to the support person? Voice Mail is a state of the art computer controlled answering system whereby each user effectively has their own password protected answering machine with enhanced capabilities, such as message forwarding. What hours do the support personnel work? Do they have other duties and/or responsibilities besides support? How many support personnel are
there? Are factory personnel, such as Product Engineers, Manufacturing Engineers, Quality Engineers, Hardware Engineers, and Software Engineers, readily available to assist the support personnel? Are the factory people on-site with the support personnel? Are the support personnel knowledgeable, have a helpful attitude, and do they "follow through" in a timely manner when they promise to do something, such as research your problem or send you something? Does it come via regular mail, UPS, or Overnight Express? Were you charged for fast delivery?

Does the Manufacturer have an Electronic Bulletin Board Service (BBS) where information such as application programs, product news, software updates, source code, bug lists, electronic mail, and conferencing are available? What baud rates are supported? How many phone lines are available? What are the hours of operation? Do you need any special brand of computer and/or modem to access it? Is there a system operator (sysop) assigned to manage it?

LITERATURE SUPPORT

Literature covers a wide selection of printed material which can assist you in the selection process. This includes items from the Manufacturer, such as Data Sheets, Data Books, and Application Notes, as well as items available at the local bookstore and/or library. Items from the local bookstore and/or library indicate not only the popularity of the manufacturers/MCsUs under consideration, but also offer unbiased opinions when written by non-manufacturer related authors.

FINALIZING THE SELECTION

As a final step to help in the selection process, build a table listing each MCU under consideration on one axis and the important attributes on the other axis. Then fill in the blanks from the manufacturer's data sheets in order to obtain a fair side-by-side comparison. Some manufacturers have pre-made comparison sheets of their MCU product line which makes this task much easier, but as with all data sheets, be sure they are up to date with current production units. Among the possible attributes are price (for the anticipated production volume, including predictions of future pricing, i.e., will the price be decreasing as you move into production?), RAM, ROM, EPROM, EEPROM, timer(s), A/D, D/A, serial ports, parallel ports (I/O control lines), bus speed (minimum/maximum), special instructions (multiply, divide, etc.), number of available interrupts, interrupt response time (time from start of interrupt to execution of the first interrupt handler instruction), package size/type (ceramic DIP or LCC, plastic 0.3" DIP or 0.6" DIP, shrink DIP (.071" pin spacing), PLCC, PQFP, EIAJ QFP, SOIC; some involve surface mount technology), power supply requirements, and any other items important to your system design. The tables at the end of this application note detail the attributes of Motorola's MCU product line.

If after all this, you still have more than one MCU on your list, consider expandability and value. What expansions in the system requirements can you predict that will be needed in possible future iterations of this product? And lastly, consider value, for if two MCUs cost the same but one offers a few more features which are not required today but would make future expansion easier for no additional cost, choose that MCU.

CUSTOM MCU SOLUTION

If there is no commercially available single-chip MCU that meets your system requirements and your anticipated production volume is high enough, you should consider using a custom CSIC MCU. In a custom CSIC MCU, you choose the core processor type and the exact peripherals needed for your system from a list of standard cells available. This gives you the benefits of a single-chip solution for slightly more cost, so the production volume must be high enough to justify it. Additionally, some manufacturers will not even start production unless the order volume is around one million units. However, if your production volume can be combined with others to reach the one million level, production could be started. Or, if the desired unit is judged to be have a broad enough market appeal, the manufacturer may proceed with production anyway because they plan to offer it as a standard product. As the design initiator you may be able to obtain an exclusivity clause whereby you have sole rights to the CSIC MCU for a specific period of time. Then the manufacturer can start marketing it to everyone.

TEAMWORK

Finally, as project leader you can do all this investigative work yourself, or you can start involving your team by assigning investigative tasks to them, such as having the software engineers evaluate the instruction sets of each MPU under consideration. By involving your team early in the decision process, you not only build team spirit, but gain individual commitment to the project via active participation. This approach undoubtedly generates some conflict, as everyone has their own opinion, but your job as project leader is to be a mediator. After listening to all opinions, it is still your choice as project leader. As in political elections, once the winner of the primary has emerged, all party members are expected to fully support the winner, and so should all project team members support the decisions of the project leader in order to ensure a successful project.

CONCLUSION

In conclusion, selecting the right MCU for your project is not an easy decision, as MCUs have become more complex devices since on-chip resources were added. And since the trend is towards more on-chip integration of off-chip resources to reduce system costs, the decision will become increasingly complex with time. This application note is not intended to make the choice for the designer, but to serve as a thought provoking guideline as to all the possible selection criteria that should be considered in this important decision process.
MOTOROLA MICROCONTROLLER COMPARISON GUIDE

Motorola maintains a comparison guide on our most popular M68HC05, M68HC11, and M68300 Families of microcontrollers (MCUs). For a copy, please contact the Motorola Literature Distribution Center nearest you and request the current MICROCONTROLLER (MCU) QUARTERLY UPDATE FOLDER, Motorola part number SG148/D.

MOTOROLA 8-BIT M68HC05 MICROCONTROLLER FAMILY

Programming Model
The M68HC05 Family of MCUs has five central processing unit (CPU) registers available to the programmer as shown in Figure 1 below.

Programming Model
The M68HC05 Family of MCUs has five central processing unit (CPU) registers available to the programmer as shown in Figure 1 below.

Figure 1. M68HC05 Programming Model

1. Length depends on the address/memory size of the individual M68HC05, e.g., a 2K memory map requires an 11-bit program counter and stack pointer as shown. Current length ranges from 11 to 14 bits (2K to 16K).

MOTOROLA 8-BIT M68HC11 MICROCONTROLLER FAMILY

Programming Model
The M68HC11 Family of MCUs has eight central processing unit (CPU) registers available to the programmer as shown in Figure 2 below.

Programming Model
The M68HC11 Family of MCUs has eight central processing unit (CPU) registers available to the programmer as shown in Figure 2 below.

Figure 2. M68HC11 Programming Model

1. Length depends on the address/memory size of the individual M68HC11, e.g., a 16K memory map requires an 11-bit program counter and stack pointer as shown. Current length ranges from 11 to 14 bits (2K to 16K).

MOTOROLA 16-/32-BIT M68300 MICROCONTROLLER FAMILY

Programming Model
The M68300 Family of MCUs has 23 central processing unit (CPU) registers available to the programmer as shown in Figures 3 and 4 below, organized into User and Supervisor models.

Programming Model
The M68300 Family of MCUs has 23 central processing unit (CPU) registers available to the programmer as shown in Figures 3 and 4 below, organized into User and Supervisor models.

Figure 3. M68300 Programming Model

1. Length depends on the address/memory size of the individual M68300, e.g., a 16K memory map requires an 11-bit program counter and stack pointer as shown. Current length ranges from 11 to 14 bits (2K to 16K).
Figure 3. M68300 User Programming Model

Figure 4. M68300 Supervisor Programming Model Supplement
DEVELOPMENT SUPPORT

Development support for Motorola MCUs is available from Motorola in the form of low cost Educational Computer Boards (ECBs), Evaluation Boards (EVBs), and Evaluation Modules (EVMs), and higher priced emulator systems with bus analysis (CDS Jewelbox and HDS-300 systems). EVBs and EVMs are on the order of hundreds of dollars, whereas emulators are on the order of thousands of dollars. Simple assemblers are available for no charge on the Freeware BBS, whereas more powerful assemblers with such features as macros, structured assembly, conditional assembly, and relocatable modules (for linking) are available for a few hundred dollars for popular PCs. Additionally, the Freeware BBS contains the most up to date information on Motorola MCUs and has a wealth of free software. Also available are literature and Application Notes on many subjects, including a title index, from Motorola's Literature Distribution Centers.

MOTOROLA FREeware ELECTRONIC BULLETIN BOARD SYSTEM

"Freeware" is the name of the electronic bulletin board system (BBS) dedicated to support Motorola Microprocessor Units (MPUs) and Microcontroller Units (MCUs). The Freeware BBS contains the most up to date information, including support software for EVMs, PCs, and MACs, development software for MCUs and MPUs, confidential electronic mail service, file downloads/uploads, distributor directory and sales offices by state, press news, development support, literature, mask set erratas, devices/packages being phased out, ECB/EVB/EVM product literature, and contest/promotion/seminar information.

Freeware is on-line 24 hours a day, everyday except for maintenance. To use the BBS, you need a 300-2400 baud modem, and a terminal or personal computer (PC) with communications software (e.g., Kermit, ProComm, etc.). Set your character format to 8-bit, no parity, 1 stop bit and dial the Freeware number (512) 891-FREE (891-3733). Press RETURN and then enter the requested information to log on. You are now a registered user. Follow the menus for the desired functions (e.g., download, upload, mail, conferences, etc.). On-line help is available.

For most up to date information, please contact the Motorola Literature Distribution Center nearest you and request the MCU FREeware brochure, Motorola part number BR568/D.

APPLICATIONS SUPPORT

Application support for Motorola products is provided by the same local salesperson and Field Application Engineer (FAE) that made the sale. If a technical question cannot be answered locally by the salesman or FAE, the FAE will contact the Factory Applications Support Group which has a highly trained staff of engineers to relentlessly pursue the answers in both hardware and software. Additionally, the Aps Group has the backup support of the entire on-site Factory staff to ensure answers to your questions.

MOTOROLA LITERATURE DISTRIBUTION CENTERS

Motorola has several Literature Distribution Centers (LDCs) worldwide with the main one located in Phoenix, Arizona, U.S.A. They carry all Motorola literature, including Data Books, Data Reference Manuals, User Manuals, Application Notes, brochures, books on Motorola products, etc. Contact your local Motorola representative or the LDC office nearest you and they will be most happy to serve you. The LDC addresses are listed on the back cover of this application note. The phone numbers are listed below or consult your local telephone directory book.

Motorola Literature Distribution Center
Phone: (602) 994-6561
Motorola Semiconductors H.K. Ltd.
Phone: 480-8333
Motorola Ltd.; European Literature Center
Phone: (908) 61 46 14
Nippon Motorola Ltd.
Phone: 03-440-3311

ABOUT MOTOROLA

Motorola was founded in 1928 by Paul V. Galvin and has continuously been in the electronics business since then. Motorola moved up to number 48 on the 1989 Fortune 500 list of the largest U.S. industrial corporations with $9.62 billion in sales and is the #3 in electronics in the U.S., behind General Electric and Westinghouse Electric. In terms of semiconductor market share, Motorola is #1 in North America and #4 in the world. Motorola'S Microcontroller Division is the world's #1 supplier of 8-bit MCUs, with more than 500 million in use around the globe. Motorola was ranked #3 by Fortune Magazine's poll of America'S Most Admired Corporations.

Motorola has consistently demonstrated product and technology leadership along with the global capability and teamwork to get the job done for the lowest possible price. Motorola is driven by Total Customer Satisfaction, which means global design and manufacturing facilities, on-time delivery, and above all, a dedication to quality that is unsurpassed in the industry. Motorola'S relentless pursuit of perfection is the key in achieving Six Sigma Quality by 1992. This translates to virtual perfection (3.4 defects per million parts). There is only one goal: zero defects in everything Motorola does.
RECENT AWARDS PRESENTED TO MOTOROLA

1990 GM Mark of Excellence Award
This award was presented by Delco Electronics for outstanding performance as a supplier in all five areas covered by GM's supplier monitoring program — quality, cost, delivery, technology and management. Currently, less than 30 suppliers have earned the GM Mark of Excellence award.

1990 Computer Design's Vendor Preference Survey reported Motorola Microcontrollers as #1 in the three measured categories of Response Time, Documentation, and Applications Support.

1989 Texas Instruments Information Technology Group Supplier Excellence Award
Presented to less than one percent of TI's suppliers worldwide for quality and on-time delivery performance.

1989 Chrysler Motor Company Pentastar Award
Presented to select group of Chrysler's suppliers worldwide for quality, price, delivery, and technology.

1989 Ford Motor Company Q1 Preferred Quality Award
A total quality supplier award; requires consistent high quality coupled with excellent pricing and delivery.

1988 Malcolm Baldrige National Quality Award
This award was created by Congress in 1987, named after the late Malcolm Baldrige, Secretary of Commerce during the Reagan Administration. It is awarded to companies that demonstrate superior company-wide management of quality processes. A panel of judges examines the quality standards in eight critical business areas for each company applying for the award. Six prizes are offered each year — two each for manufacturing companies and service companies, and two for small businesses. In the two years the award has been offered, only five companies out of a potential 12 winners have met the rigid standards required to capture the prize. In 1988, Motorola was one of only three Baldrige winners out of 66 applicants. Winners of the Malcolm Baldrige National Quality Award must wait five years before they can apply again. Therefore, Motorola will be eligible to compete again in 1993.

1988 Dataquest Supplier of the Year

1988 Texas Instruments Data Systems Group Supplier Excellence Award
Presented for demonstrated excellence in meeting Texas Instrument's requirements.

1987/1988 Bosch Group Recognition Award
Presented for quality and special performance as a supplier to the Bosch Group.

1987 Delco Electronics Award of Excellence
Presented for superior quality and delivery performance.
Reducing A/D Errors in Microcontroller Applications

Many significant benefits can be realized in an electronic product by converting analog signals into the digital domain. From drift-free signal filtering to extremely reliable signal detection, the digital domain offers a level of performance many times only approximated by its analog circuit-based counterpart. Once cost prohibitive, converting analog signals into the digital domain has become more cost effective. These decreasing costs, increasing digital semiconductor speeds, and the benefits of digital processing have contributed significantly to the increasing popularity of digital systems and to the rise of the digital system with built-in analog interfaces. One such popular system on silicon is the single-chip microcontroller unit (MCU). Now available from many manufacturers and in many forms, MCUs with resident analog interfaces like analog-to-digital converters (ADCs) and other on-chip peripherals can provide unsurpassed cost effectiveness to a product’s design. The MCU with integral ADC may easily be used to convert analog signals in the digital domain with the convenience of an already defined on-chip ADC-to-CPU interface. In addition, the MCU offers the flexibility afforded to all software-based systems.

MCUs have liberated many board-level designers from selecting, designing, and debugging microprocessor peripherals in multichip assemblies. This type of highly integrated solution is becoming more popular than the multichip solution. Consequently, it is reasonable to expect that the practicing design engineer will eventually work with an MCU-based system. Yet, despite the advantages of the MCU system, some integrated peripherals such as the ADC offer new challenges to the designer. By incorporating a wide bandwidth linear system, such as an ADC, on the same die with a high-speed digital central processor unit (CPU), ADC performance can be adversely affected. Noisy ADC readings functionally manifest themselves in a range from merely annoying and relatively benign glitches to more catastrophic hard failures. In any case, an MCU-based system does not have to be at the mercy of poor MCU/ADC performance. Fortunately, by following some fairly rudimentary systems-level guidelines in the design phase of the MCU-based product, potential ADC performance problems can be avoided.

To resolve ADC performance issues, it is necessary to understand a little about the nature of the MCU and the various areas of susceptibility of several ADC types. Although much information presented in this application note assumes that the ADC is resident on-chip with the CPU, other converter types not typically found on-chip with MCUs are discussed for those instances in which a multichip combination is encountered. The following paragraphs also apply to these less frequent hardware combinations.

ADC TYPES

Even when the ADC is available on-chip with an MCU where the unpleasant task of interfacing and debugging the ADC-to-CPU interface is done, obtaining maximum performance from the ADC requires attention to application details of the given MCU/ADC combination. The type of design precautions and applications details needed to avoid problems varies as a function of the type of ADC used. Understanding the mechanics of the given ADC is crucial to improving performance.

ADCs may be categorized into five main categories: integrating, servo, flash, successive approximation, and hybrid. Although each type has unique capabilities and traits, each has surprisingly similar points of vulnerability.

The integrating converter has appeal for applications requiring high resolution (16-bit or higher) and low cost. Because the basic converter is simply implemented (see Figure 1 (a)), hardware is minimized while high resolution is obtained. In addition, the integrating ADC may provide some noise immunity that is not feasible with higher speed designs. Although it is possible to build the integrating converter onto MCU chips (there is nothing technologically impeding such a construct), its lower speed has apparently been discouraged by MCU designers since it is currently not offered on an MCU by a major manufacturer.

Whereas the integrating converter tends to be the slowest of ADC types, the servo converter tends to have the highest resolution and fastest conversion times in its most recent advancement—the sigma-delta converter. The more traditional servo converter tries to balance the charge or voltage on an input comparator by using a feedback configuration (see Figure 1 (b)) to force slewing from a previous charge or voltage to the current input signal level applied to the other input of the same comparator. This process is followed by appropriately changing a digital counter up or down (in this form, the converter is often called a tracking converter). Before the sigma-delta variation, the servo converter was less popular than other converter types primarily due to its slew-rate limitation. Nothing about the servo converter would prevent its inclusion onto an MCU die; once BICMOS processes improve, this converter type will probably become a popular feature of future MCUs (particularly the sigma-delta variation, most of which is linear circuitry).

Although the sigma-delta variation of the servo converter provides high-resolution conversions and maintains a relatively high throughput rate, the fastest type is the flash converter. By stringing together several voltage comparators (one per desired level to be detected), conversion bandwidths in excess of 100 MHz are now quite commonplace (see Figure 1 (c)), albeit at lower resolutions (4–6 bits are common). As the input voltage is applied to one input of all the comparators, a set of
reference voltages are applied to the other comparator inputs. After a period of time has elapsed, determined primarily by the propagation delays through the comparators, the discrete-level representation of the input voltage is available at the comparator outputs. Flash conversion, although incredibly fast, requires a tremendous number of devices to implement even modest-resolution converters. In addition to the number of transistors necessary to implement each comparator, the outputs of each comparator are typically input to a combinational logic array to form a desired output code. Consequently, this converter, which consumes much silicon area when compared to other converter types, has not been widely accepted by MCU designers and users.

The fourth ADC type is the successive-approximation converter (SAC). Of all current converter types, SAC is the most popular (see Figure 1(d)). This popularity is primarily due to its applicability to smaller circuit requirements, medium to fast conversion speeds, and medium- to high-resolution applications (8–16 bits). Like other converter types, the SAC uses a differential voltage comparator to compare the input signal with a reference voltage. By performing a binary search, conversion rates of one bit per clock are possible. Because only one comparator is typically required and the output code is inherent in the conversion process, circuitry and silicon surface area are reduced when compared with other conversion methods. Although the exact implementation varies from silicon-chromium-based to charge-redistribution, this ADC is currently the most prevalent type found on MCUs.

The fifth ADC category is the hybrid converter. In this case, the term "hybrid" is not used to reference a specific implementation approach, but rather implies combining one or more AOC types to form an ADC with different performance characteristics. For example, some of the faster and higher resolution ADCs now employ a hybridized technique which utilizes flash-conversion prescaling followed by an SAC. In this case, almost instantaneous prescaling is accomplished and easily interfaced to an existing SAC design. Hybrid converters are a very viable alternative as an MCU peripheral and may find eventual popularity in MCU designs when higher resolution converters are needed.

ADC NOISE SUSCEPTIBILITY

The comparator is the cornerstone of the A/D conversion process. The ability of the comparator to announce the presence of small voltage/current differentials with large changes in its output voltage make the comparator invaluable to the A/D conversion process. Yet, this same feature also accounts for the largest potential source of ADC malfunction. Of course, degradation of the comparator’s desired action, and hence the ADC, is most usually caused by unwanted noise. Two basic characteristics of the comparator affect noise susceptibility: bandwidth and power supply connections.

Wide bandwidth comparators easily respond to noise as well as to signals. Even in the low-speed integrating converter, the accuracy of measurement is heavily contingent upon the comparator’s speed of operation. To illustrate, imagine that a
very slowly varying input signal has been applied to an input of such a comparator. For a single-slope integrating converter, the other comparator input will have a linearly increasing voltage (or other convenient shape) applied to it. As this voltage ramp increases, an independent digital counter (started at the same time the voltage ramp began) will count clock pulses provided by some timebase. When the voltage ramp finally exceeds the input voltage, the comparator will change state. If the comparator fails to respond to the voltage ramp in a timely fashion, the digital counter will register an incorrect count when compared to the results obtained by a perfectly fast comparator, implying that the response time (characterized by bandwidth) must be reasonably fast even in the slowest ADC types. Consequently, a wide bandwidth comparator will appropriately respond not only to input/reference signals but also to any other signal present at the comparator input terminals (including noise components superimposed upon the signals of interest).

The typical comparator uses some form of differential front end. The operation of the differential front end is dependent upon biasing networks that are ultimately connected to the supply terminals of the comparator. Therefore, the comparator should be considered as a five-terminal device—two differential inputs, one output, and two inputs to the biasing networks—for the purposes of designing with the ADC. The implication is that signals present at the supply terminals of an ADC, particularly the high-frequency signals typically superimposed on the power supply in digital systems, can affect comparator and ADC operation.

Due to the high bandwidth of the comparator found in ADCs, the designer of a given system should be extremely careful about the type and amount of signals allowed to reach the comparator stage of the ADC, particularly the power supply terminals. For this reason, some of the more mundane and overlooked aspects of electrical product design, such as printed circuit design and circuit interconnection, become increasingly critical to the success of the MCU/ADC system.

**APPROPRIATE DESIGN TECHNIQUES**

Most of the MCU is digital. As seen in Figure 2, a major portion of the M68HC11E9, a representative MCU, is digital circuitry. Thus, it is reasonable to assume that digital design practices will generally be employed when designing with the MCU. With an analog-based sub-system, such as the M68HC11 ADC, normally accepted digital design practices may not be sufficient to ensure satisfactory performance of the converter. As an illustration, consider noise levels normally found on the power supply of a typical high-speed HCMOS digital system. It is not unusual to find 100 mVpp broadband noise riding on top of the positive voltage rail. With a nominal 5 V HCMOS system, the resulting voltage drop, down to 4.9 V, is above the V_{OH} for HCMOS. Thus, the 100 mV signal will probably not upset circuit operation. When present in such a robust digital system (HCMOS), this 100 mV noise signal is a mere visual nuisance on the oscilloscope. Because of the theoretically infinite signal-to-noise ratio of digital gates, the presence of the 100 mV noise poses no practical threat. However, when such a noise signal is inserted into an ADC system, the results can be much more dramatic. In an 8-bit ADC system with a nominal 5 V reference, this same 100 mV noise can result in a greater than 5-bit error in the ADC reading. Thus, an MCU system utilizing an ADC assumes a different electrical character that requires application of design practices not traditionally used in the design of digital systems.

What design practices should be used? To correct or avoid a noisy ADC/MCU design, separate the noisy signals from the sensitive ones. The challenge is to design a system in which this separation is practically realized. Closer to the ideal of completely separating the noisy signals from the sensitive ones, the better. For situations where the noisy and sensitive circuits cannot be completely separated, reduce the noise coupling as much as possible. Since it is difficult to axiomatically specify how to implement both concepts in all cases, an illustration will aid understanding and provide an analogy by which individual situations may be gauged.

Motorola tests 100% of the ADCs found on their MCUs. Before any M68HC11 ADCs leave the factory, they have been tested and verified for specified ADC performance. Even so, it is possible to operate the M68HC11 in an environment that causes the M68HC11 ADC to subsequently malfunction. These two scenarios in the life of such an MCU indicate not, strictly speaking, a parts-related anomaly, but rather a significant interaction of the part's characteristics with the electrical environment.

Typically, a large contributor to malfunction is the printed circuit board (PCB) layout. Since the PCB can influence many of the circuit parametrics (reactance, voltage, etc.), the PCB layout can help or hinder ADC performance. Yet, the PCB layout is not typically done by the circuit's designer. More importantly, laying out the PCB artwork, up to and including the width and placement of traces, is often performed by people without a detailed knowledge of correct electrical circuit design practices. Many PCB designers are only concerned with ensuring that they have connected all the points connected in the schematic. Although this has its economic advantages, this can be a dangerous proposition with regard to ADC performance. Figure 3 shows an example of such a PCB layout which, although it manages to distribute the power to all of the devices, provides several potential sources of ADC/MCU performance problems.

First, the MCU/ADC is placed farthest from the power terminal, meaning that the MCU return currents will be mixed with the digital circuit currents between the MCU and the power terminal. Although the MCU may not produce large return currents in the power return, high-speed digital circuits typically do. The inductance of PCB traces at high frequencies can be significant enough to produce large noise spikes when measured between the ground pin of the MCU and the ground terminal of the board.

Second, the opamp, which buffers the signals to the ADC inputs on the MCU, is physically located close to the MCU but is electrically located in a very poor place. As with the MCU, the opamp power supply return will be corrupted with high-frequency spikes. However, the voltage drops measured...
Figure 2. MC68HC11E9 Block Diagram
between the opamp and the MCU will be even worse than those measured between the MCU and power terminal. When deciding which parts are to be jointly located on the PCB, the electrical impact of conductor distance and tolerance to any induced noise must be considered.

Third, the bypass capacitors, as shown, are ineffectual in reducing high-frequency noise on VDD. To perform the decoupling function properly, bypass capacitors should be attached as close as possible to the IC power pins they are intended to bypass. In addition, PCB trace inductance should be minimized between the leads of the capacitor and the power pins.

Figure 3 illustrates a few of the PCB-related errors that can degrade ADC performance. Specific PCB designs involving MCU/ADCs should be carefully engineered. A better PCB layout is depicted in Figure 4, which corrects the defects shown in Figure 3.
A SPECIFIC MCU WITH ADC

Other factors involving a more specific ADC system contribute to reduced ADC performance. Thus, this discussion will focus on the ADC system found on the Motorola M68HC11 Family of MCUs.

A unique implementation of an SAC, the standard M68HC11 (2 MHz bus) ADC provides a 16 μs 8-bit A/D conversion with the convenience of an on-chip MCU peripheral. The ADC is a charge-redistribution SAC. The digital-to-analog converter (DAC) is implemented with capacitors rather than the usual R-2R silicon-chromium (SiCr) thin-film resistors. Although the SiCr resistor has the advantage over the commonly used diffused resistor in improved temperature stability and tracking, laser trimming is necessary to obtain ADC accuracies compatible with even medium-resolution converters. Processing this R-2R ladder presents a challenge since trimming one resistor in the network will change the current in the previously trimmed bit, requiring an iterative trimming process. Furthermore, the R-2R ladder requires careful control of the ON resistance in the MOS switches because the switches also determine the current flow through the R-2R network. The M68HC11 capacitive DAC avoids these shortcomings. The charge-redistribution method is easily fabricated using poly-poly capacitors. No trimming of the poly capacitors or MOS switches is required to obtain medium-resolution accuracies. As an added benefit, a sample-hold function, which extends the effective conversion bandwidth of the ADC, is an inherent byproduct of the redistribution technique.

The internal operatives of the M68HC11 converter are relevant to preventing or reducing ADC errors. For converters using SiCr R-2R ladders, the impact of parametric phenomena may be different than for the M68HC11. It is necessary to understand the nature and implementation of the ADC to realize the highest performance from it. To understand the M68HC11 conversion process, a 2-bit example is presented (see Figure 5). A conversion is accomplished by a sequence of three operations. In the sample mode (see Figure 5(a)), the top plate is connected to \( V_L (0 \, \text{V}) \), and the bottom plates are connected to the input voltage, \( V_X \), resulting in a stored charge on the top plate that is proportional to the input voltage. In the hold mode (see Figure 5(b)), the top switch is then opened, and the bottom plates are connected to \( V_L \). Since the charge on the top plate is conserved, its potential goes to \( -V_X \), which is the initial voltage at the input of the comparator. The approximation mode (see Figure 5(c)), begins by testing the value of the most significant bit by raising the bottom plate of the largest capacitor to the reference voltage, \( V_H \). The equivalent circuit is now actually a voltage divider between two equal capacitances. The output of the comparator, after each capacitor is switched, determines whether the bottom plate of that capacitor will remain at \( V_H \) or be returned to \( V_L \) before the next capacitor is switched. Conversion proceeds in this manner until all bits have been determined and the result is stored in the successive-approximation register (SAR).

The following major sources of M68HC11 ADC errors controllable by external circuit parameters are discussed in the following paragraphs.

Figure 5. ADC Conversion Modes
Leakage Current on ADC Input Pin

The electrical model of an M68HC11 ADC input pin is shown in Figure 6. The problem is caused by n-channel device junction leakages at this node (there are no p-channel devices used here), which are worse at high temperatures. Consequently, the leakage current is (1) unidirectional and (2) bound by the maximum specification of 400 nA. This leakage-induced error would tend to only cause a static lowering of ADC results. To avoid leakage effects, the external circuit network feeding the ADC pin(s) should maintain impedances, which, in the presence of maximum leakage, would guarantee a maximum desired error. For example, if the maximum error (due to leakage) is desired to be ±1 LSB with a 5 V reference voltage, then the maximum source impedance (resistance) feeding this pin should be 50 kΩ = (19.5 mV/400 nA).

Charge Time on Sample Capacitor

By lengthening the resistance-capacitance (RC) time constant, comprised of the source resistance feeding the ADC pin and the DAC capacitance evidenced at the pin during the sample mode, errors may result. However, given the size of the DAC input capacitance, the size of the source resistance necessary to induce these RC time-constant errors will probably be inundated by the effects of pin leakage described previously.

VDD/VSS and Input Terminal Noise

The differential comparator used in the M68HC11 ADC derives its power from VDD and VSS, the power pins that supply the rest of the M68HC11 (see Figure 7). The M68HC11,

**Figure 6. Electrical Model of an M68HC11 ADC Input Pin**

**Figure 7. MC68HC11 ADC in Sample Mode**
when considered with respect to ADC performance, is a source of noise, partially due to the waveshape and harmonics associated with square waves. In addition, the complex relationship between the primary M68HC11 clock and related noise voltages are further complicated by dependance of the M68HC11 upon many software combinations, each sufficiently changing the noise characteristics emanating from the M68HC11. Therefore, ADC performance degradation, which is linked to noise generated on VDD/VSS by the M68HC11, can often appear related to execution of specific software combinations. As established earlier, this is due to the ADC wide bandwidth comparator.

NOTE
Because the M68HC11 ADC uses a very wide bandwidth comparator capable of responding to noise components in excess of 20 MHz, it must be guarded against unwanted noise at its input terminals and VDD/VSS pins.

The reference to input terminal noise must be distinguished between noise externally superimposed on the input signal lines that is measured between a system reference and a given input signal (occurs from capacitive coupling between high-impedance ADC inputs and noisy signal sources or electro-magnetic interference) and voltage differentials experienced by different comparator inputs when referenced to each other. The importance of input terminal noise in this context is the presence of non-common-mode differential noise between the biasing networks in the comparator and the input lines. If, under noisy conditions, the same noise is presented to an input to the comparator and one of the supply (or other input) terminals, the common-mode rejection ratio (CMRR) capabilities of the comparator may prevent performance perturbations; whereas, noise presented to either terminal, with respect to system ground, may cause havoc (see Wide Bandwidth Input Signals). Efforts should be made to ensure that noise, if it cannot be reduced further, is also seen by the other comparator inputs to take advantage of the CMRR. Of the ADC error sources, this is one of the most challenging to control in a practical and effective manner.

Wide Bandwidth Input Signals
A certain way to disrupt ADC function is to give the wide bandwidth comparator something to respond to other than the input signal of interest. By designing the electronics feeding the ADC inputs to pass input signals having frequencies that range from DC to purple, ADC problems are usually guaranteed. Thus, this fourth area is a common source of ADC malfunction.

Other Error Sources
Although occurring less frequently and more subtly, other error sources can also impact ADC performance: rate of conversion requests to a particular channel and interchannel charge-sharing. These sources and an estimate of the impact on a given M68HC11 system are presented in detail in M68HC11RM/AD, M68HC11 Reference Manual.

REAL-WORLD EXAMPLE
When discussing the mechanics of noise phenomena in MCU/ADC systems, it is very difficult to understand how large the noise problem is, how well it is expected to respond to corrective action, and how closely the analysis matches the real world. To help resolve these problems, an actual troubleshooting session involving an M68HC11-based assembly is presented.

The subject assembly, an industrial controller, is a typical MCU/Controller module utilizing the M68HC11 in expanded multiplexed (CPU) mode. The customer designed the program memory to expand to 32K x 8, RAM to 2K x 8, an external address decoder, some additional digital I/O lines, and analog buffers feeding the ADC inputs. Built on a six-layer PCB, the assembly had the benefit of separate ground and voltage planes, and was designed to be placed in a Faraday shield providing electromagnetic compatibility. This assembly was designed without the aid of any of the concepts presented in this application note. Understandably, the customer was having difficulty with ADC performance.

The Problem
Functionally, the ADC noise problem manifested itself as an extreme shutdown condition in the final product. Since this assembly provides control to industrial equipment, conditions sensed by this controller could indicate dangerous conditions, which must be dealt with by severe and swift action, including functional shutdown of the controlled equipment. To achieve the safest response times and largest safety margins to such stimuli, the software designers of this system required 64 μs continuous conversions (>15 kHz sampling frequency). Once they were run through part of the designer’s algorithm, the conversion results could not deviate more than ±2 counts from the actual system ADC measurements. The M68HC11 was selected for this application because of its high level of integration as well as the ±1 LSB 8-bit ADC performance. Errors many times this specification were encountered in the application. Unfortunately, evaluation of the extent of the ADC errors concerned only functional operation of the assembly and manual inspection of ADC values read with an in-circuit emulator, making the problem more serious. An attempt by the hardware engineers to reduce the noise by changing the bypassing scheme yielded no apparent change in the pattern of product shutdown. In this case, the lack of quantitative data convinced the engineers that they had no control over the problem, diverting attention from the actual cause. When dealing with these types of problems, always instrument the problem correctly—that is, ensure measuring techniques used to observe the malfunction follow these guidelines:

1. Quantify the A/D conversion process with regard to frequency of occurrence and magnitude of error.
2. Ensure that the measurements are with sufficient resolution so that minute improvements or degradations in performance may be monitored and evaluated.
3. Ensure that the number of observed conversions are similar to product usage or are statistically significant to allow inference from the measured sample to actual product operation.
Had the assembly been properly monitored, an improvement in ADC performance with the different bypassing scheme would have been evident (see Figure 8). These two histograms display ranges of A/D conversion values on the vertical axis and the hit rate (percent of total readings landing within the boundaries of the selected ADC reading range) on the horizontal axis. As shown, there was approximately a 3% improvement in the number of correct A/D conversions with new bypassing. To detect these changes, the EPROM on the controller PCB was probed with a fairly simple logic analyzer. The logic analyzer was then configured to trigger on accesses to a location in memory containing the results of A/D conversions. By utilizing the simple statistics options given by the analyzer, each quantitative improvement in ADC performance was observed.

**The Perfect Circuit**

After sufficiently instrumenting the offending assembly, the next step is to attempt to duplicate ideal operating conditions for the MCU/ADC. Since every M68HC11 is 100% tested for ADC performance before leaving the factory, in the absence of externally induced failure, the M68HC11 should maintain factory performance given identical operating conditions. By operating the M68HC11 in near perfect conditions, the engineer learns if the failure is or is not parts related. The motivating factor for this case, however, concerned Vpp noise (spaced in time at approximately the E-clock rate of the M68HC11) found when measuring VDD at the pins of the M68HC11. Given what is known about the ADC comparator, it was best for system performance to reduce this VDD noise as much as possible. The noise was reduced by isolating the power bussing to the M68HC11 only. The PCB foil was cut to VDD, VSS, and VRL.

### Figure 8. Effect of Bypassing Only

**Table A**: ADC Reading (Hex)

<table>
<thead>
<tr>
<th>ADC Reading (Hex)</th>
<th>Freq.</th>
<th>Conditions: 0.1 μF Ceramic Bypass</th>
<th>Total Samples = 8192</th>
<th>M68HC11 Industrial Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-90</td>
<td>0 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91-A9</td>
<td>98 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99-A0</td>
<td>1 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1-A8</td>
<td>0 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9-B0</td>
<td>0 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1-B8</td>
<td>0 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9-C0</td>
<td>0 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1-FF</td>
<td>0 %</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### (a) No Bypassing

### (b) Improved Bypassing

**NOTE**: Readings are rounded to nearest 1% value. Columns with 0% and a grey bar imply >0% and <0.5%.
and \( V_{NH} \) leading to the M68HC11. Discrete wires were then run directly to an external laboratory-grade power supply. With this configuration, measurements were taken as before. The results of these measurements are shown in Figure 9. As the graph shows, an improvement was made over the non-bypassed assembly. Instead of a 5% error in the ADC readings, less than 0.5% of the readings were outside of the expected range. Also evident in Figure 9 is the presence of full-scale errors as before. At this point, a bypass capacitor was soldered between the M68HC11 \( V_{DD} \) and \( V_{SS} \) pins. The resulting measurements, shown in Figure 10, are an apparent improvement over the previous non-bypassed assembly. However, due to the granularity of the measurement reported by the logic analyzer, it cannot be stated quantitatively how much the bypassing improved the condition. Further manipulation of the bypassing network failed to improve the readings in a discernible manner.

At this point, only power distribution busses had been manipulated to reduce ADC errors. Another part of the ADC circuit manipulated to yield some improvement was the linear portion interfacing the MCU to the various input signals. Consisting of 324-type opamps operated at unity gain, this linear buffer provided a low-impedance source for the ADC input multiplexer. Although not usually considered a wideband op-amp, it proved too wideband for this system. Most data coming from the devices feeding the 324 buffers were slowly varying DC or signals with frequencies below 500 Hz. Yet, the full bandwidth of signals allowed by the buffer passed unaltered to

<table>
<thead>
<tr>
<th>ADC READING (HEX)</th>
<th>FREQ</th>
<th>CONDITIONS:</th>
<th>SEPARATE MCU POWER SUPPLY</th>
<th>TOTAL SAMPLES = 16384</th>
<th>M68HC11 INDUSTRIAL CONTROLLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-80</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81-98</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99-A0</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1-A8</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9-B0</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1-B8</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9-C0</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1-FF</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(EXPANDED SCALE) PERCENT OCCURRENCE

NOTE: Readings are rounded to nearest 1% value. Columns with 0% and a grey bar imply >0% and <0.5%.

**Figure 9. Separate MCU Power Supply**

<table>
<thead>
<tr>
<th>ADC READING (HEX)</th>
<th>FREQ</th>
<th>CONDITIONS:</th>
<th>SEPARATE MCU POWER SUPPLY &amp; BYPASS</th>
<th>TOTAL SAMPLES = 16384</th>
<th>M68HC11 INDUSTRIAL CONTROLLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-80</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81-98</td>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99-A0</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1-A8</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9-B0</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1-B8</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9-C0</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1-FF</td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(EXPANDED SCALE) PERCENT OCCURRENCE

NOTE: Readings are rounded to nearest 1% value. Columns with 0% and a grey bar imply >0% and <0.5%.

**Figure 10. Separate MCU Power Supply with 0.1 \( \mu \)F Bypass**
the ADC inputs. This manipulation violated a design guideline that urges the designer to tailor the bandwidth of each ADC channel to the bandwidth of the input signal. By properly filtering the input to the ADC, frequencies that may prove troublesome if left unfiltered will not be allowed to pass to the ADC input. To test the affect of this guideline on this specific industrial controller, 0.01 μF capacitors were soldered to the ADC input pins at the M68HC11. The measurements taken with this configuration (see Figure 11) showed significant improvement. As shown in Figure 11, there were still occasional occurrences of out-of-spec ADC readings.

In the absence of other guidelines, the only choice left to achieve specified ADC performance was to refine the implementation of the existing guidelines. The second guideline, duplicate ideal operating conditions, is usually the most likely candidate for improvement. One of the corollaries to duplicating ideal operating conditions is reducing unwanted interaction between adjacent circuit segments. In this case, VDD noise had not been completely eradicated. Rather than inserting a local IC regulator for just the M68HC11, an alternative method of VDD isolation was attempted: a series diode with VDD forming a peak detector with the bypass cap. Out-of-spec ADC errors were totally eliminated (see Figure 12(a)). To check the thoroughness of this last circuit fix, the range of sensitivity for the ADC result range of interest was changed on the logic analyzer. By changing the range to show values between (94)16 and (96)16, inclusively, the ±1 LSB spec could be observed directly. The results of this measurement run are shown in Figure 12(b).

**SUMMARY**

The highly integrated MCU can be a cost-effective design tool. With the breadth of MCU choices available to the circuit designer these days, analog circuit functions may now often be implemented by MCUs with integral ADCs. By following the practical guidelines presented in this application note during the design phase, the MCU-based product design using the on-chip ADC can achieve its full cost-effective potential.

<table>
<thead>
<tr>
<th>ADC READING (HEX)</th>
<th>FREQ.</th>
<th>CONDITIONS: 0.01 μF CAPACITOR AT ADC PIN</th>
<th>M68HC11 INDUSTRIAL CONTROLLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-80</td>
<td>0 %</td>
<td>TOTAL SAMPLES = 16384</td>
<td></td>
</tr>
<tr>
<td>81-98</td>
<td>100 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99-A0</td>
<td>0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1-A8</td>
<td>0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9-B0</td>
<td>0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1-B8</td>
<td>0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9-C0</td>
<td>0 %</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1-FF</td>
<td>0 %</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Readings are rounded to nearest 1% value. Columns with 0% and a grey bar imply >0% and <0.5%.

Figure 11. Capacitor on ADC Pin
<table>
<thead>
<tr>
<th>ADC READING (HEX)</th>
<th>FREQ.</th>
<th>CONDITIONS: DIODE IN V&lt;sub&gt;DD&lt;/sub&gt;</th>
<th>M68HC11 INDUSTRIAL CONTROLLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-80</td>
<td>0%</td>
<td>TOTAL SAMPLES = 167.9K</td>
<td></td>
</tr>
<tr>
<td>81-98</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99-A0</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1-A8</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9-B0</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1-B8</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9-C0</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1-FF</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(EXPANDED SCALE) PERCENT OCCURRENCE

(a) Diode in V<sub>DD</sub>

<table>
<thead>
<tr>
<th>ADC READING (HEX)</th>
<th>FREQ.</th>
<th>CONDITIONS: FINAL CIRCUIT</th>
<th>M68HC11 INDUSTRIAL CONTROLLER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-93</td>
<td>0%</td>
<td>TOTAL SAMPLES = 167.9K</td>
<td></td>
</tr>
<tr>
<td>94-96</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>97-A0</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1-A8</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9-B0</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1-B8</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9-C0</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1-FF</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(EXPANDED SCALE) PERCENT OCCURRENCE

(b) Tightened ADC Range

NOTE: Readings are rounded to nearest 1% value. Columns with 0% and a grey bar imply >0% and <0.5%.

Figure 12. V<sub>DD</sub> Diode and Tightened ADC Range
MC68HC11 Bootstrap Mode

Prepared by: Jim Sibigtroth
Mike Rhoades
John Langan

INTRODUCTION

M68HC11 MCUs have a bootstrap mode that allows a user-defined program to be loaded into the internal random access memory (RAM) by way of the serial communications interface (SCI); the M68HC11 then executes this loaded program. The loaded program can do anything a normal user program can do as well as anything a factory test program can do because protected control bits are accessible in bootstrap mode. Although the bootstrap mode is a single-chip mode of operation, expanded mode resources are accessible because the mode control bits can be changed while operating in the bootstrap mode.

This application note explains the operation and application of the M68HC11 bootstrap mode. Although the basic concepts associated with this mode are quite simple, the more subtle implications of these functions require careful consideration. Useful applications of this mode are overlooked due to an incomplete understanding of the bootstrap mode. Also, common problems associated with the bootstrap mode could be avoided by a more complete understanding of its operation and implications.

Topics included in this application note are as follows:
- Basic operation of the M68HC11 bootstrap mode
- General discussion of bootstrap mode uses
- Detailed explanation of on-chip bootstrap logic
- Detailed explanation of bootstrap firmware
- Bootstrap firmware vs. EEPROM security
- Incorporating the bootstrap mode into a system
- Driving bootstrap mode from another M68HC11
- Driving bootstrap mode from a personal computer
- Common bootstrap mode problems
- Variations for specific versions of M68HC11
- Commented listings for selected M68HC11 bootstrap ROMs

BASIC BOOTSTRAP MODE

This section describes only basic functions of the bootstrap mode. Other functions of the bootstrap mode are described in detail in the remainder of this application note.

When an M68HC11 is reset in bootstrap mode, the reset vector is fetched from a small internal read-only memory (ROM) called the bootstrap ROM or (boot ROM). The firmware program in this boot ROM then controls the bootloading process. First, the on-chip SCI is initialized. The first character received ($FF) determines which of two possible baud rates should be used for the remaining characters in the download operation. Next, a binary program is received by the SCI system and is stored in RAM. Finally, a jump instruction is executed to pass control from the bootloader firmware to the user's loaded program. Bootstrap mode is useful both at the component level and after the MCU has been embedded into a finished user system.

At the component level, Motorola uses the bootstrap mode to control a monitored burn-in program for the on-chip electrically erasable programmable read-only memory (EEPROM). Units to be tested are loaded into special circuit boards that each hold fifty MCUs. These boards are then placed in burn-in ovens. Driver boards outside the ovens download an EEPROM exercise and diagnostic program to all fifty MCUs in parallel. The MCUs under test independently exercise their internal EEPROM and monitor programming and erase operations. This technique could be utilized by an end user to load program information into the EPROM or EEPROM of an M68HC11 before it is installed into an end product. As in the burn-in setup, many M68HC11s can be gang programmed in parallel. This technique can also be used to program the EPROM of finished products after final assembly.

Motorola also uses bootstrap mode for programming target devices on the M68HC11EVM Evaluation Modules. Because bootstrap mode is a privileged mode like special test, the EEPROM-based configuration register (CONFIG) can be programmed using bootstrap mode on the EVM.

The greatest benefits from bootstrap mode are realized by designing the finished system so that bootstrap mode can be used after final assembly. The finished system need not be a single-chip mode application for the bootstrap mode to be useful because the expansion bus can be enabled after resetting the MCU in bootstrap mode. Allowing this capability requires almost no hardware or design cost and the addition of this capability is invisible in the end product until it is needed.

The ability to control the embedded processor through downloaded programs is achieved without the disassembly and chip-swapping usually associated with such control. This mode provides an easy way to load non-volatile memories such as EEPROM with calibration tables or to program the application firmware into a one-time programmable (OTP) MCU after final assembly.

Another powerful use of bootstrap mode in a finished assembly is for final test. Short programs can be downloaded to check parts of the system, including components and circuitry external to the embedded MCU. If any problems appear during product development, diagnostic programs can be downloaded to find the problems, and corrected routines can be downloaded and checked before incorporating them into the main application program.
Bootstrap mode can also be used to interactively calibrate critical analog sensors. Since this calibration is done in the final assembled system, it can compensate for any errors in discrete interface circuitry and cabling between the sensor and the analog inputs to the MCU. Note that this calibration routine is a downloaded program that does not take up space in the normal application program.

**BOOTSTRAP MODE LOGIC**

In the MC68HC11 very little logic is dedicated to the bootstrap mode: Thus, this mode adds almost no extra cost to the MCU system. The biggest piece of circuitry for bootstrap mode is the small boot ROM. This ROM is 192 bytes in the original MC68HC11A8, but some of the newest members of the M68HC11 Family have as much as 448 bytes to accommodate added features. Normally, this boot ROM is present in the memory map only when the MCU is reset in the bootstrap mode to prevent interference with the user’s normal memory space. The enable for this ROM is controlled by the read boot ROM (RBOOT) control bit in the highest priority interrupt (HPRI0) register. The RBOOT bit can be written by software whenever the MCU is in special test or special bootstrap modes; when the MCU is in normal modes, RBOOT reverts to zero and becomes a read-only bit. All other logic in the MCU would be present whether or not there was a bootstrap mode.

Figure 1 shows the composite memory map of the MC68HC711E9 in its four basic modes of operation, including bootstrap mode. The active mode is determined by the mode A (MDA) and special mode (SMOD) control bits in the HPRI0 control register. These control bits are in turn controlled by the state of the mode A (MODA) and mode B (MOBD) pins during reset. Table 1 shows the relationship between the state of these pins during reset, the selected mode, and the state of the MDA, SMOD, and RBOOT control bits. Refer to the composite memory map and Table 1 for the following discussion.

The MDA control bit is determined by the state of the MODA pin as the MCU leaves reset. MDA selects between single-chip and expanded operating modes. When MDA is zero, a single-chip mode is selected, either normal single-chip or special bootstrap mode. When MDA is one, an expanded mode is selected, either normal expanded mode or special test mode.

The SMOD control bit is determined by the inverted state of the MODB pin as the MCU leaves reset. SMOD controls whether a normal mode or a special mode is selected. When SMOD is zero, one of the two normal modes is selected, either normal single-chip or normal expanded mode. When SMOD is one, one of the two special modes is selected, either special bootstrap mode or special test mode. When either special mode is in effect (SMOD = 1), certain privileges are in effect — i.e., the ability to write to the mode control bits and fetching the reset and interrupt vectors from $8FFxx rather than $FFxx.

The alternate vector locations are achieved by simply driving address bit A14 low during all vector fetches if SMOD = 1. For special test mode, the alternate vector locations assure that the reset vector can be fetched from external memory space so the test system can control MCU operation. In special bootstrap mode, the small boot ROM is enabled in the memory map by RBOOT = 1 so the reset vector will be fetched from this ROM and the bootloader firmware will control MCU operation.

RBOOT is reset to one in bootstrap mode to enable the small boot ROM. In the other three modes, RBOOT is reset to zero to keep the boot ROM out of the memory map. While in special test mode, SMOD = 1; which allows the RBOOT control bit to be written to one by software to enable the boot ROM for testing purposes.

**BOOT ROM FIRMWARE**

The main program in the boot ROM is the bootloader, which is automatically executed as a result of resetting the MCU in bootstrap mode. Some newer versions of the M68HC11 Family have additional utility programs that can be called from a downloaded program. One utility is available to program EPROM or OTP versions of the M68HC11. A second utility allows the contents of memory locations to be uploaded to a host computer. In the MC68HC711K4 boot ROM, a section of code is used by Motorola for stress testing the on-chip EEPROM. These test and utility programs are similar to self-test ROM programs in other MCUs except that the boot ROM does not use valuable space in the normal memory map.

Bootstrap firmware is also involved in an optional EEPROM security function on some versions of the M68HC11. This EEPROM security feature prevents a software pirate from seeing what is in the on-chip EEPROM. The secured state is invoked by programming the no security (NOSEC) EEPROM bit in the CONFIG register. Once this NOSEC bit is programmed to zero, the MCU will ignore the mode A pin and always come out of reset in normal single-chip mode or special bootstrap mode, depending on the state of the mode B pin. Normal single-chip mode is the usual way a secured part would be used. Special bootstrap mode is used to disengage the security function (only after the contents of EEPROM and RAM have been erased). Refer to the M68HC11RM/AD, *M68HC11 Reference Manual* for additional information on the security mode and complete listings of the boot ROMs that support the EEPROM security functions.

**AUTOMATIC SELECTION OF BAUD RATE**

The bootloader program in the MC68HC711E9 accommodates either of two baud rates. The higher of these baud rates (7812 baud at a 2-MHz E-clock rate) is used in systems that operate from a binary frequency crystal such as $2^{33}$ Hz (8.389 MHz). At this crystal frequency the baud rate is 8192 baud which was used extensively in automotive applications based on the MC6801 MCU. The second baud rate available to the M68HC11 bootloader is 1200 baud at a 2-MHz E-clock rate. Some of the newest versions of the M68HC11 accommodate other baud rates using the same differentiation technique explained here. Refer to the reference numbers in square brackets in Figure 2 during the following explanation.

---

**Table 1. Mode Selection Summary**

<table>
<thead>
<tr>
<th>Input Pins</th>
<th>Control Bits in HPRI0</th>
<th>Mode Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODB</td>
<td>MODA</td>
<td>RBOOT</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
NOTE: Software can change some aspects of the memory map after reset.

Figure 1. MC68HC711E9 Composite Memory Map
Figure 2 shows how the bootloader program differentiates between the default baud rate (7812 baud at a 2-MHz E-clock rate) and the alternate baud rate (1200 baud at a 2-MHz E-clock rate). The host computer sends an initial $FF character, which is used by the bootloader to determine the baud rate that will be used for the downloading operation. The top half of Figure 2 shows normal reception of $FF. Receive data samples at [1] detect the falling edge of the start bit and then verify the start bit by taking a sample at the center of the start bit time. Samples are then taken at the middle of each bit time [2] to reconstruct the value of the received character (all ones in this case). A sample is then taken at the middle of the stop bit time as a framing check (a one is expected) [3]. Unless another character immediately follows this $FF character, the receive data line will idle in the high state as shown at [4].

The bottom half of Figure 2 shows how the receiver will incorrectly receive the $FF character that is sent from the host at 1200 baud. Because the receiver is set to 7812 baud, the receive data samples are taken at the same times as in the upper half of Figure 2. The start bit at 1200 baud [5] is 6.5 times as long as the start bit at 7812 baud [6].

Samples taken at [7] detect the falling edge of the start bit and verify it as a logic zero. Samples taken at the middle of what the receiver thinks is the first five bit times [8] detect logic zeros. The sample taken at the middle of what the receiver thinks is bit 5 [9] may detect either a zero or a one because the receive data has a rising transition at about this time. The samples for bits 6 and 7 detect ones, causing the receiver to think the received character was $C0 or $E0 [10] at 7812 baud instead of the $FF which was sent at 1200 baud. The stop bit sample detects a one as expected [11], but this detection is actually in the middle of bit 0 of the 1200 baud $FF character. The SCI receiver is not confused by the rest of the 1200 baud $FF character because the receive data line is high [12] just as it would be for the idle condition. If a character other than $FF is sent as the first character, an SCI receive error could result.

**MAIN BOOTLOADER PROGRAM**

Figure 3 is a flowchart of the main bootloader program in the MC68HC711E9. This bootloader demonstrates the most important features of the bootloaders used on all M68HC11 family members. For complete listings of other M68HC11 versions refer to Listings 3–8 at the end of this application note, and appendix B of the M68HC11RM/AD, _M68HC11 Reference Manual_.

The reset vector in the boot ROM points to the start [1] of this program. The initialization block [2] establishes starting conditions and sets up the SCI and port D. The stack pointer is set because there are push and pull instructions in the bootloader program. The X index register is pointed at the start of the register block ($1000) so indexed addressing can be used. Indexed addressing takes one less byte of ROM space than extended instructions, and bit manipulation instructions are not available in extended addressing forms. The port D wire-OR mode (DWOM) bit in the serial interface control register (SPCR) is set to configure port D for wired-OR operation to minimize potential conflicts with external systems that use the PD1/TxD pin as an input. The baud rate for the SCI is initially set to 7812 baud at a 2-MHz E-clock rate but can automatically switch to 1200 baud based on the first character received. The SCI receiver and transmitter are enabled. The receiver is required by the bootloading process, and the transmitter is used to transmit data back to the host computer for optional verification. The last item in the initialization is to set an intercharacter delay constant used to terminate the download when the host computer stops sending data to the MC68HC711E9. This delay constant is stored in the timer output compare 1 (TOC1) register, but the on-chip timer is not used in the bootloader program. This example illustrates the extreme measures used in the bootloader firmware to minimize memory usage. However such measures are not usually considered good programming technique because they are misleading to someone trying to understand the program.

After initialization, a break character is transmitted [3] by the SCI. By connecting the TxD pin to the RxD pin (with a pullup because of port D wired-OR mode), this break will be received as a $00 character and cause an immediate jump [4] to the start of the on-chip EEPROM ($B600 in the MC68HC711E9). This feature is useful to pass control to a program in EEPROM essentially from reset. Refer to _COMMON BOOTSTRAP MODE PROBLEMS_ before using this feature.

If the first character is received as $FF, the baud rate is assumed to be the default rate (7812 baud at a 2-MHz E-clock rate). If $FF was sent at 1200 baud by the host, the SCI will receive the character as $E0 or $C0 because of the baud rate mismatch, and the bootloader will switch to 1200 baud [5] for the rest of the download operation. When the baud rate is switched to 1200 baud, the delay constant used to monitor the
intercharacter delay must also be changed to reflect the new character time.

At [6], the Y index register is initialized to $0000 to point to the start of on-chip RAM. The index register Y is used to keep track of where the next received data byte will be stored in RAM. The main loop for loading begins at [7].

The number of data bytes in the downloaded program can be any number between zero and 512 bytes (the size of on-chip RAM). This procedure is called 'variable-length download' and is accomplished by ending the download sequence when an idle time of at least four character times occurs after the last character to be downloaded. In M68HC11 Family members which have 256 bytes of RAM, the download length is fixed at exactly 256 bytes plus the leading $FF character.

The intercharacter delay counter is started [8] by loading the delay constant from TOC1 into the X index register. The 19-E-cycle wait loop is executed repeatedly until either a character is received [9] or the allowed intercharacter delay time expires [10]. For 7812 baud, the delay constant is 10,241 E cycles (539 X 19 E cycles per loop). Four character times at 7812 baud is 10,240 E cycles (baud prescaler of 4 X baud divider of 4 X 16 internal SCI clocks/bit time X 10 bit times/character X 4 character times). The delay counter is not started until after the first character ($FF) is received.

To terminate the bootloading sequence and jump to the start of RAM without downloading any data to the on-chip RAM, simply send $FF and nothing else. This feature is similar to the jump to EEPROM at [4] except the $FF causes a jump to the start of RAM. This procedure requires that the RAM has been loaded with a valid program since it would make no sense to jump to a location in uninitialized memory.

After receiving a character, the downloaded byte is stored in RAM [11]. The data is transmitted back to the host [12] as an indication that the download is progressing normally. At [13], the RAM pointer is incremented to the next RAM address. If the RAM pointer has not passed the end of RAM, the main download loop (from [7] to [14]) is repeated.

When all data has been downloaded, the bootloader goes to [16] because of an intercharacter delay timeout [10] or because the entire 512-byte RAM has been filled [15]. At [16], the X and Y index registers are set up for calling the PROGRAM utility routine, which saves the user from having to do this in a downloaded program. The PROGRAM utility is fully explained in EPROM PROGRAMMING UTILITY. The final step of the bootloader program is to jump to the start of RAM [17], which starts the user's downloaded program.

Figure 3. MC68HC711E9 Bootloader Flowchart
UPLOAD UTILITY

The UPLOAD utility subroutine transfers data from the MCU to a host computer system over the SCI serial data link. Note that M68HC11 versions that support EEPROM security do not include this utility. Verification of EPROM contents is one example of how the UPLOAD utility could be used. Before calling this program, the Y index register is loaded (by user firmware) with the address of the first data byte to be uploaded. If a baud rate other than the current SCI baud rate is to be used for the upload process, the user’s firmware must also write to the BAUD register. The UPLOAD program sends successive bytes of data out the SCI transmitter until a reset is issued (the upload loop is infinite). For a complete commented listing of the UPLOAD utility, refer to Listings at the back of this application note.

EPROM PROGRAMMING UTILITY

The EPROM programming utility is one way of programming data into the internal EPROM of the MC68HC711E9 MCU. An external 12-V programming power supply is required to program on-chip EPROM. The simplest way to use this utility program is to bootstrap a three-byte program consisting of a single jump instruction to the start of the PROGRAM utility program ($BF00). The bootloader program sets the X and Y index registers to default values before jumping to the downloaded program (see [16] at the bottom of Figure 3). When the host computer sees the $FF character, data to be programmed into the EPROM is sent, starting with the character for location $D000. After the last byte to be programmed is sent to the MC68HC711E9 and the corresponding verification data is returned to the host, the programming operation is terminated by resetting the MCU.

The number of bytes to be programmed, the first address to be programmed, and the programming time can be controlled by the user if values other than the default values are desired.

To understand the detailed operation of the EPROM programming utility, refer to Figure 4 during the following discussion. Figure 4 is composed of three interrelated parts. The upper-left portion shows the flowchart of the PROGRAM utility running in the boot ROM of the MCU. The upper-right portion shows the flowchart for the user-supplied driver program running in the host computer. The lower portion of Figure 4 is a timing sequence showing the relationship of operations between the MCU and the host computer. Reference numbers in the flowcharts in the upper half of Figure 4 have matching numbers in the lower half to help the reader relate the three parts of the figure.

The shaded area [1] refers to the software and hardware latency in the MCU leading to the transmission of a character (in this case, the $FF). The shaded area [2] refers to a similar latency in the host computer (in this case, leading to the transmission of the first data character to the MCU).

The overall operation begins when the MCU sends the first character ($FF) to the host computer, indicating that it is ready for the first data character. The host computer sends the first data byte [3] and enters its main loop. The second data character is sent [4], and the host then waits [5] for the first verify byte to come back from the MCU.

After the MCU sends $FF [8], it enters the WAIT1 loop [9] and waits for the first data character from the host. When this character is received [10] the MCU programs it into the address pointed to by the Y index register. When the programming time delay is over, the MCU reads the programmed data, transmits it to the host for verification [11], and returns to the top of the WAIT1 loop to wait for the next data character [12]. Because the host previously sent the second data character, it is already waiting in the SCI receiver of the MCU. Steps [13], [14], and [15] correspond to the second pass through the WAIT1 loop.

Back in the host, the first verify character is received, and the third data character is sent [6]. The host then waits for the second verify character [7] to come back from the MCU. The sequence continues as long as the host continues to send data to the MCU. Since the WAIT1 loop in the PROGRAM utility is an indefinite loop, reset is used to end the process in the MCU after the host has finished sending data to be programmed.

ALLOWING FOR BOOTSTRAP MODE

Since bootstrap mode requires very few connections to the MCU, it is easy to design systems that accommodate the bootstrap mode. Bootstrap mode is useful for diagnosing or repair systems that have failed due to changes in the CONFIG register or failures of the expansion address/data buses, (rendering programs in external memory useless). Bootstrap mode can also be used to load information into the EPROM or EEPROM of an M68HC11 after final assembly of a module. Bootstrap mode is also useful for performing system checks and calibration routines. The following paragraphs explain system requirements for use of bootstrap mode in a product.

MODE SELECT PINS: It must be possible to force the MODA and MOBD pins to logic zero, which implies that these two pins should be pulled up to VDD through resistors rather than being tied directly to VDD. If mode pins are connected directly to VDD it is not possible to force a mode other than the one the MCU is hard wired for. It is also good practice to use pulldown resistors to VSS rather than connecting mode pins directly to VSS because it is sometimes a useful debug aid to attempt reset in modes other than the one the system was primarily designed for. Physically, this requirement sometimes calls for the addition of a test point or a wire connected to one or both mode pins. Mode selection only uses the mode pins while RESET is active.

RESET: It must be possible to initiate a reset while the mode select pins are held low. In systems where there is no provision for manual reset, it is usually possible to generate a reset by turning power off and back on.

RxD PIN: It must be possible to drive the PDO/RxD pin with serial data from a host computer (or another MCU). In many systems, this pin is already used for SCI communications; thus no changes are required.

In systems where the PDO/RxD pin is normally used as a general-purpose output, a serial signal from the host can be connected to the pin without resulting in output driver conflicts. It may be important to consider what the existing logic will do with the SCI serial data instead of the signals that would have been produced by the PDO pin. In systems where the PDO pin is normally used as a general-purpose input, the driver circuit
Figure 4. Host and MCU Activity during EPROM PROGRAM Utility
that drives the PD0 pin must be designed so that the serial data can override this driver, or the driver must be disconnected during the bootstrap download. A simple series resistor between the driver and the PD0 pin solves this problem as shown in Figure 5. The serial data from the host computer can then be connected to the PD0/RxD pin, and the series resistor will prevent direct conflict between the host driver and the normal PD0 driver.

![Figure 5. Preventing Driver Conflict](image)

**Figure 5. Preventing Driver Conflict**

**TxD PIN:** The bootloader program uses the PD1/TxD pin to send verification data back to the host computer. To minimize the possibility of conflicts with circuitry connected to this pin, port D is configured for wire-OR mode by the bootloader program during initialization. Since the wire-OR configuration prevents the pin from driving active high levels, a pullup resistor to VDD is needed if the TxD signal is used.

In systems where the PD1/TxD pin is normally used as a general-purpose output, there are no output driver conflicts. It may be important to consider what the existing logic will do with the SCI serial data instead of the signals that would have been produced by the PD1 pin.

In systems where the PD1 pin is normally used as a general-purpose output, the driver circuit that drives the PD1 pin must be configured so that the PD1/TxD pin driver in the MCU can override this driver. A simple series resistor between the driver and the PD1 pin can solve this problem. The TxD pin can then be configured as an output, and the series resistor will prevent direct conflict between the internal TxD driver and the external driver connected to PD1 through the series resistor.

**OTHER:** The bootloader firmware sets the DWOM control bit, which configures all port D pins for wire-OR operation. During the bootloading process, all port D pins except the PD1/TxD pin are configured as high-impedance inputs. Any port D pin that is normally used as an output should have a pullup resistor so it does not float during the bootloading process.

**DRIVING BOOT MODE FROM ANOTHER M68HC11**

A second M68HC11 system can easily act as the host to drive bootstrap loading of an M68HC11 MCU. This method is used to examine and program nonvolatile memories in target M68HC11s in Motorola EVMs. The following hardware and software example will demonstrate this and other bootstrap mode features.

The schematic in Figure 6 shows the circuitry for a simple EPROM duplicator for the MC68HC711E9. The circuitry is built in the wire-wrap area of an M68HC11EVBU Evaluation Board to simplify construction. The schematic shows only the important portions of the EVBU circuitry to avoid confusion. To see the complete EVBU schematic, refer to the M68HC11EVBU/D, M68HC11EVBU Universal Evaluation Board User's Manual.

The default configuration of the EVBU must be changed to make the appropriate connections to the circuitry in the wire-wrap area and to configure the master MCU for bootstrap mode. A fabricated jumper must be installed at J6 to connect the XTAL output of the master MCU to the wire-wrap connector P5, which has been wired to the EXTAL input of the target MCU. Cut traces that short across J8 and J9 must be cut on the solder side of the printed circuit board to disconnect the normal SCI connections to the RS232 level translator (U4) of the EVBU. The J8 and J9 connections can easily be restored at a later time by installing fabricated jumpers on the component side of the board. A fabricated jumper must be installed across J3 to configure the master MCU for bootstrap mode.

One MC68HC711E9 is first programmed by other means with a desired 12K-byte program in its EPROM and a small duplicator program in its EEPROM. Alternately, the ROM program in an MC68HC711E9 can be copied into the EPROM of a target MC68HC711E9 by programming only the duplicator program into the EVBU of the master MC68HC711E9. The master EVBU is installed in the EVBU at socket U3. A blank MC68HC711E9 to be programmed is placed in the socket in the wire-wrap area of the EVBU (U6).

With the Vpp power switch off, power is applied to the EVBU system. As power is applied to the EVBU, the master MCU (U3) comes out of reset in bootstrap mode. Target MCU (U6) is held in reset by the PB7 output of master MCU (U3). The PB7 output of U3 is forced to zero when U3 is reset. The master MCU will later release the reset signal to the target MCU under software control. The RxD and TxD pins of the target MCU (U6) are high-impedance inputs while U6 is in reset so they will not affect the TxD and RxD signals of the master MCU (U3) while U3 is coming out of reset. Since the target MCU is being held in reset with MODA and MODB at zero, it is configured for the EPROM emulation mode, and PB7 is the output enable signal for the EPROM data I/O pins. Pullup resistor R7 causes the port D pins including RxD and TxD, to remain in the high-impedance state so they do not interfere with the RxD and TxD pins of the master MCU as it comes out of reset.

As U3 leaves reset, its mode pins select bootstrap mode so the bootloader firmware begins executing. A break is sent out the TxD pin of U3. Pullup resistor R10 and resistor R9 cause the break character to be seen at the RxD pin of U3. The bootloader performs a jump to the start of EEPROM in the master MCU (U3) and starts executing the duplicator program. This sequence demonstrates how to use bootstrap mode to pass control to the start of EEPROM after reset.

The complete listing for the duplicator program in the EEPROM of the master MCU is provided in Listing 1. The duplicator program in EEPROM clears the DWOM control bit to change port D (thus, TxD) of U3 to normal driven outputs. This configuration will prevent interference due to R9 when TxD from the target MCU (U6) becomes active. Series resistor R9 demonstrates how TxD of U3 can drive RxD of U3 and later TxD of U6 can drive RxD of U3 without a destructive conflict between the TxD output buffers.
NOTE: Only the most important portions of EVBU circuitry are shown.

Figure 6. MCU to MCU EPROM Duplicator Schematic
As the target MCU (U6) leaves reset, its mode pins select bootstrap mode so the bootloader firmware begins executing. A break is sent out the TxD pin of U6. At this time, the TxD pin of U3 is at a driven high so R9 acts as a pullup resistor for TxO of the target MCU (U6). The break character sent from U6 is received by U3 so the duplicator program that is running in the EEPROM of the master MCU knows that the target MCU is ready to accept a bootloaded program.

The master MCU sends a leading $FF character to set the baud rate in the target MCU. Next, the master MCU passes a three-instruction program to the target MCU and pauses so the bootstrap program in the target MCU will stop the loading process and jump to the start of the downloaded program. This sequence demonstrates the variable-length download feature of the MC68HC711E9 bootloader.

The short program downloaded to the target MCU clears the DwOM bit to change its TxO pin to a normal driven CMOS output and jumps to the EPROM programming utility in the bootstrap ROM of the target MCU.

Note that the small downloaded program did not have to set up the SCI or initialize any parameters for the EPROM programming process. The bootstrap software that ran prior to the loaded program left the SCI turned on and configured in a way that was compatible with the SCI in the master MCU (the duplicator program in the master MCU also did not have to set up the SCI for the same reason). The programming time and starting address for EPROM programming in the target MCU were also set to default values by the bootloader software before jumping to the start of the downloaded program.

Before the EPROM in the target MCU can be programmed, the Vpp power supply must be available at the XIRQ/VppE pin of the target MCU. The duplicator program running in the master MCU monitors this voltage (for presence or absence — not level) at PE7 through resistor divider R14 - R15. The PE7 input was chosen because the internal circuitry for port E pins can tolerate voltages slightly higher than VDD; therefore resistors R14 and R15 are less critical. No data to be programmed is passed to the target MCU until the master MCU senses that Vpp has been stable for about 200 ms.

When Vpp is ready, the master MCU turns on the red LED and begins passing data to the target MCU. EPROM PROGRAMMING UTILITY explains the activity as data is sent from the master MCU to the target MCU and programmed into the EPROM of the target. The master MCU in the EVBU corresponds to the HOST in the programming utility description, and the "PROGRAM utility in MCU" is running in the bootstrap ROM of the target MCU.

Each byte of data sent to the target is programmed and then the programmed location is read and sent back to the master for verification. If any byte fails, the red and green LEDs are turned off, and the programming operation is aborted. If the entire 12K bytes are programmed and verified successfully, the red LED is turned off, and the green LED is turned on to indicate success. The programming of all 12K bytes takes about 30 sec.

After a programming operation, the Vpp switch (S2) should be turned off before the EVBU power is turned off.
### Listing 1. MCU to MCU Duplicator Program

```
1  ******************************************************
2  * 68HC711E9 Duplicator Program for AN1060         
3  ******************************************************
4  
5  *****  
6  * Equates - All reg addr except INIT are 2-digit  
7  * for direct addressing                           
8  *****  
9  103D  INIT EQU $103D  RAM, Reg mapping            
10  0028  SPCR EQU $28  DWOM in bit-5                 
11  0004  PORTB EQU $04  Red LED = bit-1, Grn = bit-0 
12  * Reset of prog socket = bit-7                   
13  0080  RESET EQU $10000000  
14  0002  RED EQU $00000010                             
15  0001  GREEN EQU $00000001                           
16  000A  PORTA EQU $0A  Vpp Sense in bit-7, 1=ON      
17  002E  SCSR EQU $2E  SCI status register            
18  002F  TDRE, TC, RDRF, IDLE; OR, NF, FE, -         
19  0080  RDRF EQU $10000000  SCI data register       
20  0020  SCRD EQU $2F  
21  00F0  PROGRAM EQU $8000  EPROM prog utility in boot ROM  
22  D000  EPSTRT EQU $0000  Starting address of EPROM   
23  0000  
24  B600  ORG $8600  Start of EEPROM                  
25  0000  
26  ******************************************************
27  
28  B600  7F103D  BEGIN  
29  B603  8604  CLR INIT  
30  B605  9728  LDA #504  Pattern for DWOM off, no SPI  
31  B607  8680  STAA SPCR  
32  B609  9704  LDA #RESET                            
33  B60B  13E20FC  BRCCLR SCSR RDRF WT4BRK  Loop till char received 
34  B60F  86FF  LDA #$FF  Leading char for bootload ...  
35  B611  97FF  STAA SCRD  
36  B613  CEB675  LDX #BLPROG  
37  B616  BD53  BLOOP  
38  B618  BCB67D  
39  B61B  26F9  BNE BLOOP  
40  B61C  
41  
42  * Delay for about 4 char times to allow boot related  
43  * SCI communications to finish before clearing      
44  * Rx related flags                                  
45  B61D  CED6A7  
46  B620  09  LDX #1703  
47  B621  26FD  DLYLP  
48  B623  962E  LDAA SCSR  
49  B625  962F  LDAA SCRD  
50  8680  
51  9728  
52  
53  13E20FC  WT4FF  BRCCLR SCSR RDRF WT4FF  Wait for RDRF  
54  B62B  962F  LDAA SCRD  
55  B62D  CED000  EQU #EDSTRT  
56  
57  B627  13E20FC  WT4FF  BRCCLR SCSR RDRF WT4FF  Wait for RDRF  
58  B62B  962F  LDAA SCRD  
59  B62D  CED000  EQU #EDSTRT  
60  
61  * Handle turn-on of Vpp                            
62  B630  18CE523D  WT4VPP  LDX #1053  Delay counter (about 200ms)  
63  B634  150402  BCLR PORTB000  Turn off RED LED  
64  B637  960A  DLYLP2  
65  B639  2A5  BSET PORTB RED  
66  B63B  140402  BSET PORTB RED  
67  B63E  1809  DEY  
68  B640  26F5  BNE DLYLP2  
69  962F  
70  962E  
71  * Vpp has been stable for 200ms                    
72  18CED000  
73  B642  18CED000  
74  B646  8D23  BSR SEND1  
75  B648  8C0000  DATALP  
76  B64A  2702  BEQ VERF  
77  B64D  8DC1  BSR SEND1  
78  B64F  13E20FC  VERF  BRCCLR SCSR RDRF VERF  Wait for Rx ready  
79  B653  962F  LDAA SCRD  
80  B655  18A100  
81  B658  2705  BSR VERFOK  
82  B65A  150403  
83  B65D  2007  
84  B65F  
85  B65F  1808  VERFOK  
86  B661  2685  BNE DATALP  
87  B665  2685  BNE DATALP  
88  495  
```
Listing 1. MCU to MCU Duplicator Program

80 B663 140401 BSET PORTB GREEN Grn LED ON
82 B666
83 B666 150482 DUNPRG BCLR PORTB (RESET+RED) Red OFF, apply reset
84 B669 20FE BRA * Done so just hang
85 B66B
86 ******************************************************
87 * Subroutine to get & send an SCI char. Also
88 * advances pointer (X).
89 ******************************************************
90 B66B A600 SEND1 LDAA C,X Get a character
91 B666 B71028 STAA $1028 Pattern for DWOM off, no SPI
92 B671 972F STAA SCDR Send character
93 B673 08 INX Advance pointer
94 B674 39 RTS ** Return **
95 96
97 ******************************************************
98 * Program to be boot loaded to target '711E9
99 ******************************************************
100 B675 8604 BLPROG LDAA #804 Pattern for DWOM off, no SPI
101 B677 B71028 STAA $1028 Turns off DWOM in target MCU
102 * NOTE: Can't use direct addressing in target MCU because
103 * regs are located at $1000.
104 B67A 7EBF00 JMP PROGRAM Jumps to EPROM prog routine
105 B67D ENDBPR EQU *

Symbol Table:
Symbol Name Value Def. # Line Number Cross Reference
BEGIN B600 *00029
BLLOOP B616 *00038 00040
BLPROG B675 *00099 00037
DATALP B648 *00068 00079
DLYLP B620 *00046 00047
DLYLP2 B637 *00059 00063
DUNPRG B666 *00083 00076
ENDBPR B67D *00104 00039
EPSTR D000 *00023 00055 00066
GREEN 0001 *00015 00075 00081
INIT 103D *00009 00029
PORTB 0004 *00011 00033 00058 00061 00075 00081 00083
PORTE 000A *00016 00059
PROGRAM BF00 *00022 00103
RDRO 0020 *00020 00034 00053 00071
RED 0002 *00014 00058 00061 00075 00083
RESET 0080 *00013 00032 00083
SCDR 002F *00021 00036 00049 00054 00072 00092
SCSR 002E *00017 00034 00048 00053 00071 00091
SEND1 B66B *00090 00038 00067 00070
SPCR 0028 *00010 00031
TDR 0080 *00019 00091
TDYLP B66D *00091 00051
VERF B64F *00071 00069 00071
VERFOK B65F *00078 00074
WT4BRK B60B *00034 00034
WT4FF B627 *00053 00053
WT4VPP B630 *00057 00060

Errors: None
Labels: 28
Last Program Address: $B67C
Last Storage Address: $0000
Program Bytes: $007D 125
Storage Bytes: $0000 0

496
DRIVING BOOT MODE FROM A PERSONAL COMPUTER

In this example, a personal computer is used as the host to drive the bootloader of an MC68HC711E9. An M68HC11 EVBU is used for the target MC68HC711E9. A large program is transferred from the personal computer into the EPROM of the target MC68HC711E9.

HARDWARE: Figure 7 shows a small modification to the EVBU to accommodate the 12-V (nominal) EPROM programming voltage. The XIRQ pin is connected to a pulled resistor, two jumpers, and the 60-pin connectors, P4 and P5. The object of the modification is to isolate the XIRQ pin and then connect it to the programming power supply. Carefully cut the trace on the solder side of the EVBU as indicated in Figure 7. This disconnects the pullup resistor RNI0 from XIRQ but leaves P4-18, P5-18, and jumpers J7 and J14 connected so the EVBU can still be used for other purposes after programming is done. Remove any fabricated jumpers from J7 and J14. The EVBU normally has a jumper at J7 to support the trace function.

Figure 8 shows a small circuit that is added to the wire-wrap area of the EVBU. The three-terminal jumper allows the XIRQ line to be connected to either the programming power supply or to a substitute pullup resistor for XIRQ. The 100-ohm resistor is a current limiter to protect the 12-V input of the MCU. The resistor and LED connected to P5 pin 9 (port C bit 0) is an optional indicator that lights when programming is complete.

SOFTWARE: BASIC was chosen as the programming language due to its readability and availability in parallel versions on both the IBM® PC and the Macintosh™. The program demonstrates several programming techniques for use with an M68HC11 and is not necessarily intended to be a finished, commercial program. For example, there is very little error checking, and the user interface is very elementary. A complete listing of the BASIC program is included in Listing 2 with moderate comments. The following paragraphs include a more detailed discussion of the program as it pertains to communicating with and programming the target MC68HC711E9. Lines 25–45 initialize and define the variables and array used in the program. Changes to this section would allow for other programs to be downloaded.

1. IBM is a trademark of International Business Machines.
2. Macintosh is a trademark of Apple Computers, Inc.

Figure 7. Isolating EVBU XIRQ Pin

Figure 8. PC to MCU Programming Circuit

497
Lines 50–95 read in the small bootloader from DATA statements at the end of the listing. The source code for this bootloader is presented in the DATA statements. The bootloaded code makes port C bit 0 low, initializes the X and Y registers for use by the EPROM programming utility routine contained in the boot ROM, and then jumps to that routine. The hexadecimal values read in from the DATA statements are converted to binary values by a subroutine. The binary values are then saved as one string (BOOTCODE$).

The next long section of code (lines 97–1250) reads in the S-records from an external disk file (in this case, BUF34.S19), converts them to integer, and saves them in an array. The techniques used in this section show how to convert ASCII S-records to binary form that can be sent (bootloaded) to an M68HC11.

This S-record translator only looks for the S1 records that contain the actual object code. All other S-record types are ignored.

When an S1 record is found (line 1000–1024), the next two characters form the hex byte giving the number of hex bytes to follow. This byte is converted to integer by the same subroutine that converted the bootloaded code from the DATA statements. This BYTECOUNT is adjusted by subtracting 3, which accounts for the address and checksum bytes and leaves just the number of object-code bytes in the record.

Starting at line 1100, the two-byte (four-character) starting address is converted to decimal. This address is the starting address for the object-code bytes to follow. An index into the CODE$ array is formed by subtracting the base address initialized at the start of the program from the starting address for this S-record.

A FOR-NEXT loop starting at line 1130 converts the object-code bytes to decimal and saves them in the CODE$ array. When all the object-code bytes have been converted from the current S-record, the program loops back to find the next S1 record.

A problem arose with the BASIC programming technique used. The draft versions of this program tried saving the object-code bytes directly as binary in a string array. This caused “Out of Memory” or “Out of String Space” errors on both a 2M Macintosh and a 640K PC. The solution was to make the array an integer array and perform the integer-to-binary conversion on each byte as it is sent to the target part.

The one compromise made to accommodate both Macintosh and PC versions of BASIC is in lines 1500 and 1505. Use line 1500 and comment out line 1505 if the program is to be run on a Macintosh and, conversely, use line 1505 and comment out line 1500 if a PC is used.

After the COM port is opened, the code to be bootloaded is modified by adding the $FF to the start of the string. $FF synchronizes the bootloader in the MC68HC711E9 to 1200 baud. The entire string is simply sent to the COM port by PRINTing the string. This is possible since the string is actually queued in BASIC’s COM buffer, and the operating system takes care of sending the bytes out one at a time. The M68HC11 echoes the data received for verification. No automatic verification is provided, though the data is printed to the screen for manual verification.

Once the MCU has received this bootloaded code, the bootloader automatically jumps to it. The small bootloaded program in turn includes a jump to the EPROM programming routine in the boot ROM.

Refer to the previous explanation of the EPROM PROGRAMMING UTILITY for the following discussion. The host system sends the first byte to be programmed through the COM port to the SCI of the MCU. The SCI port on the MCU buffers one byte while receiving another byte, increasing the throughput of the EPROM programming operation by sending the second byte while the first is being programmed.

When the first byte has been programmed, the MCU reads the EPROM location and sends the result back to the host system. The host then compares what was actually programmed to what was originally sent. A message indicating which byte is being verified is displayed in the lower half of the screen. If there is an error, it is displayed at the top of the screen.

As soon as the first byte is verified, the third byte is sent. In the meantime, the MCU has already started programming the second byte. This process of verifying and queuing a byte continues until the host finishes sending data. If the programming is completely successful, no error messages will have been displayed at the top of the screen. Subroutines follow the end of the program to handle some of the repetitive tasks. These routines are short, and the commenting in the source code should be sufficient explanation.

MODIFICATIONS: This example programmed version 3.4 of the BUFFALO monitor into the EPROM of an MC68HC711E9; the changes to the BASIC program to download some other program are minor. The necessary changes are as follows:

1. In line 30, the length of the program to be downloaded must be assigned to the variable "CODESIZE$".
2. Also in line 30, the starting address of the program is assigned to the variable "ADRSTART$".
3. In line 9570, the start address of the program is stored in the third and fourth items in that DATA statement in hexadecimal.
4. If any changes are made to the number of bytes in the boot code in the DATA statements in lines 9500–9580, then the new count must be set in the variable "BOOTCOUNT$" in line 25.

OPERATION: Configure the EVBU for boot mode operation by putting a jumper at J3. Ensure that the trace command jumper at J7 is not installed because this would connect the 12-V programming voltage to the OC5 output of the MCU.

Connect the EVBU to its DC power supply. When it is time to program the MCU EPROM, turn on the 12-V programming power supply to the new circuitry in the wire-wrap area.

Connect the EVBU serial port to the appropriate serial port on the host system. For the Macintosh, this is the modem port with a modem cable. For the MS-DOS computer, it is connected to COM1 with a "straight through" or modem cable. Power up the host system and start the BASIC program. If the program has not been compiled, this is accomplished from within the appropriate BASIC compiler or interpreter. Power up the EVBU.

Answer the prompt for filename with either a [RETURN] to accept the default shown or by typing in a new filename and pressing [RETURN].

The program will inform the user that it is working on converting the file from S-records to binary. This process will take from 30 sec to a few minutes, depending on the computer.
A prompt reading, “Comm port open?” will appear at the end of the file conversion. This is the last chance to ensure that everything is properly configured on the EVBU. Pressing [RETURN] will send the bootcode to the target MC68HC711E9. The program then informs the user that the bootload code is being sent to the target, and the results of the echoing of this code are displayed on the screen.

Another prompt reading “Programming is ready to begin. Are you?” will appear. Turn on the 12-V programming power supply and press [RETURN] to start the actual programming of the target EPROM.

### Listing 2. BASIC Program for Personal Computer

```
1 '**********************************************************************************************
2 |
3 ' E9BVF.BAS - A PROGRAM TO DEMONSTRATE THE USE OF THE BOOT MODE ON THE HC11 BY PROGRAMMING AN MC68HC711E9 WITH BUFFALO 3.4
4 |
5 |
6 ' REQUIRES THAT THE S-RECORDS FOR BUFFALO (BUF34.S19) BE AVAILABLE IN THE SAME DIRECTORY OR FOLDER
7 |
8 ' THIS PROGRAM HAS BEEN RUN BOTH ON A MS-DOS COMPUTER USING QUICKBASIC 4.5 AND ON A MACINTOSH USING QUICKBASIC 1.0.
9 |
10 '**********************************************************************************************
25 H$ = "0123456789ABCDEF" 'STRING TO USE FOR HEX CONVERSIONS
30 DEFINT B, I: CODESIZE% = 8192: ADDRSTART = 57344!
35 BOOTCOUNT = 25 'NUMBER OF BYTES IN BOOT CODE
40 DIM CODE$(CODESIZE%) 'BUFFALO 3.4 IS 8K BYTES LONG
45 BOOTCODE$ = " " 'INITIALIZE BOOTCODE$ TO NULL
49 REM ***** READ IN AND SAVE THE CODE TO BE BOOT LOADED *****
50 FOR I = 1 TO BOOTCOUNT '# OF BYTES IN BOOT CODE
55 READ Q$
60 A$ = MID$(Q$, 1, 11)
65 GOSUB 7000 'CONVERTS HEX DIGIT TO DECIMAL
70 TEMP = 16 * X 'HANG ON TO UPPER DIGIT
75 A$ = MID$(Q$, 2, 1)
80 GOSUB 7000
85 TEM$ = TEMP + X
90 BOOTCODE$ = BOOTCODE$ + CHR$(TEMP) 'BUILD BOOT CODE
95 NEXT I
96 REM ***** S-RECORD CONVERSION STARTS HERE *****
97 FILNAM$="BUF34.S19" 'DEFAULT FILE NAME FOR S-RECORDS
100 CLS
105 PRINT "Filename.ext of S-record file to be downloaded (";FILNAM$;":" )
107 INPUT Q$
110 IF Q$<>"" THEN FILNAM$=Q$
115 OPEN FILNAM$ FOR INPUT AS #1
120 PRINT : PRINT "Converting ": FILNAM$; " to binary...
999 REM ***** SCANS FOR 'Sl' RECORDS *****
1000 GOSUB 6000 'GET 1 CHARACTER FROM INPUT FILE
1010 IF A$ = 1 THEN 1250 'FLAG IS EOF FLAG FROM SUBROUTINE
1020 IF A$ <> "S" THEN 1000
1022 GOSUB 6000
1024 IF A$ = 1 THEN 1000
1029 REM ***** SI RECORD FOUND, NEXT 2 HEX DIGITS ARE THE BYTE COUNT *****
1030 GOSUB 6000
1040 GOSUB 7000 'RETURNS DECIMAL IN X
1050 BYTECOUNT = 16 * X 'ADJUST FOR HIGH NIBBLE
1060 GOSUB 6000
1070 GOSUB 7000 'ADJUST FOR ADDRESS + CHECKSUM
1080 BYTECOUNT = BYTECOUNT + X
1090 BYTECOUNT = BYTECOUNT - 3 'ADD LOW NIBBLE
1099 REM ***** NEXT 4 HEX DIGITS BECOME THE STARTING ADDRESS FOR THE DATA *****
1100 GOSUB 6000 'GET FIRST NIBBLE OF ADDRESS
1102 GOSUB 7000 'CONVERT TO DECIMAL
```
Listing 2. BASIC Program for Personal Computer

1104 ADDRESS= 4096 * X
1106 GOSUB 6000 'GET NEXT NIBBLE
1108 GOSUB 7000
1110 ADDRESS= ADDRESS+ 256 * X
1112 GOSUB 6000
1114 GOSUB 7000
1116 ADDRESS= ADDRESS+ 16 * X
1118 GOSUB 6000
1120 GOSUB 7000
1122 ADDRESS= ADDRESS+ X
1124 ARRAYCNT = ADDRESS-ADRSTART 'INDEX INTO ARRAY
1129 REM ***** CONVERT THE DATA DIGITS TO BINARY AND SAVE IN THE ARRAY *****
1130 FOR I = 1 TO BYTECOUNT
1140 GOSUB 6000
1150 GOSUB 7000
1160 Y = 16 * X
1170 GOSUB 6000
1180 GOSUB 7000
1190 Y = Y + X
1200 CODE%(ARRAYCNT) = Y
1220 NEXT I
1230 GOTO 1000
1232 CLOSE 1
1499 REM ***** DUMP BOOTLOAD CODE TO PART *****
1500 'OPEN "R",#2,"COM:1200,N,8,1" 'Macintosh COM statement
1510 OPEN "COM:1200,N,8,1,CS0,CS0,CS0,R$" FOR RANDOM AS #2 'DOS COM statement
1512 WHILE LOC(2) >0 'FLUSH INPUT BUFFER
1513 GOSUB 8020
1514 WEND
1515 PRINT : PRINT "Sending bootload code to target part..."
1520 AS = CHR$(255) + BOOTCODE$ 'ADD HEX FF TO SET BAUD RATE ON TARGET HC11
1530 GOSUB 6500
1540 PRINT
1550 FOR I = 1 TO BOOTCOUNT '# OF BYTES IN BOOT CODE BEING ECHOED
1560 GOSUB 8000
1564 K=ASC(BS):GOSUB 8500
1565 PRINT "Character #": I; " received = "; HX$ 
1570 NEXT I
1575 CLS
1579 WHILE LOC(2) > 0 'FLUSH INPUT BUFFER
1580 GOSUB 8020
1590 WEND
1600 XMT = 0; RCV = 0 'POINTERS TO XMIT AND RECEIVE BYTES
1610 AS = CHR$(CODE%(XMT))
1620 GOSUB 6500 'SEND FIRST BYTE
1625 FOR I = 1 TO CODESIZE% - 1 'ZERO BASED ARRAY 0 -> CODESIZE-1
1630 AS = CHR$(CODE%(I))
1635 GOSUB 6500 'SEND IT
1640 GOSUB 8000 'GET BYTE FOR VERIFICATION
1650 RCV = I - 1
1655 K=CODE%(RCV):GOSUB 8500
1660 LOCATE 10,1:PRINT "Verifying byte ": I; " 
1664 IF CHR$(CODE%(RCV)) = BS THEN 1670
1665 K=CODE%(RCV):GOSUB 8500
1666 LOCATE 10,1:PRINT "Byte ": I; " ", " - Sent "; HX$;
1668 K=ASC(BS):GOSUB 8500
1669 PRINT " Received "; HX$;
1670 NEXT I
1680 GOSUB 8000 'GET BYTE FOR VERIFICATION
1690 RCV = CODESIZE% - 1
1700 LOCATE 10,1:PRINT "Verifying byte ": CODESIZE%; " 
1710 IF CHR$(CODE%(RCV)) = BS THEN 1720
1715 K=CODE%(RCV):GOSUB 8500
1714 LOCATE 10,1:PRINT "Byte ": CODESIZE%; " ", " - Sent "; HX$;
1720 LOCATE 8, 1: PRINT : PRINT "Done!!!!";
1725 PRINT " 
1730 END
COMMON BOOTSTRAP MODE PROBLEMS

It is not unusual for a user to encounter problems with bootstrap mode because it is new to many users. By knowing some of the common difficulties, the user can avoid them or at least recognize and quickly correct them.

Reset conditions vs. conditions as bootloaded program starts.

It is common to confuse the reset state of systems and control bits with the state of these systems and control bits when a bootloaded program in RAM starts. Between these times, the bootloader program is executed, which changes the states of some systems and control bits.

- The SCI system is initialized and turned on (Rx and Tx).
- The SCI system has control of the P0D and PD1 pins.
- Port D outputs are configured for wire-OR operation.
- The stack pointer is initialized to the top of RAM.
- Time has passed (two or more SCI character times).
- Timer has advanced from its reset count value.

Users also forget that bootstrap mode is a special mode; thus privileged control bits are accessible, and write protection for some registers is not in effect. The bootstrap ROM is in the memory map. The DISR bit in the TEST1 control register is set, which disables resets from the COP and clock monitor systems. Since bootstrap is a special mode, these conditions can be changed by software. The bus can even be switched from single-chip mode to expanded mode to gain access to external memories and peripherals.

Connecting Rx&D to Vss does not cause the SCI to receive a break — To force an immediate jump to the start of EEPROM, the bootstrap firmware looks for the first received
### Table 2. Summary of Boot-ROM-Related Features

<table>
<thead>
<tr>
<th>MCU Part #</th>
<th>BOOT ROM Revision @SBFD1</th>
<th>Mask Set I.D. @SBFD2,3</th>
<th>MCU Type I.D. @SBFD4,5</th>
<th>Security</th>
<th>Download Length</th>
<th>JMP on BRK or $001</th>
<th>JMP to RAM2</th>
<th>Default RAM Location</th>
<th>EPROM PROGRAM Utility</th>
<th>UPLOAD4 Utility</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC68HC11A0</td>
<td>—</td>
<td>—</td>
<td>Mask Set #</td>
<td>—</td>
<td>256</td>
<td>$8600</td>
<td>$0000</td>
<td>$0000–FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68HC11A1</td>
<td>—</td>
<td>—</td>
<td>Mask Set #</td>
<td>—</td>
<td>256</td>
<td>$8600</td>
<td>$0000</td>
<td>$0000–FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68HC11A8</td>
<td>—</td>
<td>—</td>
<td>Mask Set #</td>
<td>—</td>
<td>256</td>
<td>$8600</td>
<td>$0000</td>
<td>$0000–FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68SEC11A8</td>
<td>—</td>
<td>—</td>
<td>Mask Set #</td>
<td>Yes</td>
<td>256</td>
<td>$8600</td>
<td>$0000</td>
<td>$0000–FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68HC11D3</td>
<td>$00</td>
<td>Mask Set #</td>
<td>$11D3</td>
<td>—</td>
<td>0–192</td>
<td>$F000-ROM</td>
<td>—</td>
<td>$0040–FF</td>
<td>—</td>
<td>Yes</td>
<td>6</td>
</tr>
<tr>
<td>MC68HC711D3</td>
<td>$24 (B)</td>
<td>$0000</td>
<td>$0000</td>
<td>—</td>
<td>0–192</td>
<td>$F000-EPROM</td>
<td>—</td>
<td>$0040–FF</td>
<td>—</td>
<td>Yes</td>
<td>6</td>
</tr>
<tr>
<td>MC68HC811E2</td>
<td>—</td>
<td>$0000</td>
<td>$0000</td>
<td>—</td>
<td>0–192</td>
<td>$8600</td>
<td>$0000</td>
<td>$0000–FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68SEC811E2</td>
<td>—</td>
<td>—</td>
<td>$0000</td>
<td>—</td>
<td>0–192</td>
<td>$8600</td>
<td>$0000</td>
<td>$0000–FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68HC11E0</td>
<td>—</td>
<td>Mask Set #</td>
<td>$E9E9</td>
<td>—</td>
<td>0–512</td>
<td>$8600</td>
<td>—</td>
<td>$0000–1FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68HC11E1</td>
<td>—</td>
<td>Mask Set #</td>
<td>$E9E9</td>
<td>—</td>
<td>0–512</td>
<td>$8600</td>
<td>—</td>
<td>$0000–1FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68HC11E9</td>
<td>—</td>
<td>Mask Set #</td>
<td>$E9E9</td>
<td>—</td>
<td>0–512</td>
<td>$8600</td>
<td>—</td>
<td>$0000–1FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68SEC11E9</td>
<td>—</td>
<td>Mask Set #</td>
<td>$E95C</td>
<td>Yes</td>
<td>0–512</td>
<td>$8600</td>
<td>—</td>
<td>$0000–1FF</td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>MC68HC711E9</td>
<td>$41 (A)</td>
<td>$0000</td>
<td>$71E9</td>
<td>—</td>
<td>0–512</td>
<td>$8600</td>
<td>—</td>
<td>$0000–1FF</td>
<td>Yes</td>
<td>Yes</td>
<td>6</td>
</tr>
<tr>
<td>MC68HC11F1</td>
<td>$42 (B)</td>
<td>$0000</td>
<td>$F1F1</td>
<td>—</td>
<td>0–1024</td>
<td>$F000</td>
<td>—</td>
<td>$0000–3FF</td>
<td>—</td>
<td>—</td>
<td>6, 8</td>
</tr>
<tr>
<td>MC68HC11K4</td>
<td>$30 (0)</td>
<td>Mask Set #</td>
<td>$044B</td>
<td>0–768</td>
<td>$0080</td>
<td>—</td>
<td>$0080–37F</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>6, 8</td>
</tr>
<tr>
<td>MC68HC711K4</td>
<td>$42 (B)</td>
<td>$0000</td>
<td>$744B</td>
<td>0–768</td>
<td>$0080</td>
<td>—</td>
<td>$0080–37F</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>6, 8</td>
</tr>
</tbody>
</table>

**NOTES:**

1. By sending $00 or a break as the first SCI character after reset in bootstrap mode, a jump (JMP) is executed to the address in this table rather than doing a download. Unless otherwise noted, this address is the start of EEPROM. Tying Rx0 to Tx0 and using a pullup resistor from Tx0 to VDD will cause the SCI to see a break as the first received character.

2. If $55 is received as the first character after reset in bootstrap mode, a jump (JMP) is executed to the start of on-chip RAM rather than doing a download. This $55 character must be sent at the default baud rate (7812 baud @ E = 2MHz).

   For devices with variable-length download, the same effect can be achieved by sending $FF and no other SCI characters. After four SCI character times, the download terminates, and a jump (JMP) to the start of RAM is executed.

   The jump to RAM feature is only useful if the RAM was previously loaded with a meaningful program.

3. A callable utility subroutine is included in the bootstrap ROM of the indicated versions to program bytes of on-chip EPROM with data received via the SCI.

4. A callable utility subroutine is included in the bootstrap ROM of the indicated versions to upload contents of on-chip memory to a host computer via the SCI.

5. The complete listing for this bootstrap ROM may be found in the M68HC11RM/AD, M68HC11 Reference Manual.

6. The complete listing for this bootstrap ROM is included in this application note.

7. Due to the extra program space needed for EEPROM security on this device, there are no pseudo-vectors for SCI, SPI, PAIF, PAOVF, TOF, OC5F, or OC4F interrupts.

8. This bootloader extends the automatic software detection of baud rates to include 9600 baud at 2-MHz E-clock rate.
character to be $00$ (or break). The data reception logic in the SCI looks for a one-to-zero transition on the RxD pin to synchronize to the beginning of a receive character. If the RxD pin is tied to ground, no one-to-zero transition occurs. The SCI transmitter sends a break character when the bootloader firmware starts, and this break character can be fed back to the RxD pin to cause the jump to EEPROM. Since TxD is configured as an open-drain output, a pullup resistor is required.

An $SFF$ character is required before data is loaded into RAM — The initial character (usually $SFF$) that sets the download baud rate is often forgotten.

Original M68HC11 versions required exactly 256 bytes to be downloaded to RAM — Even users that know about the 256 bytes of download data sometimes forget the initial $SFF$ that makes the total number of bytes required for the entire download operation equal to $256 + 1$ or 257 bytes.

Variable-length download — When on-chip RAM surpassed 256 bytes, the time required to serially load this many characters became more significant. The variable-length download feature allows shorter programs to be loaded without sacrificing compatibility with earlier fixed-length download versions of the bootloader. The end of a download is indicated by an idle RxD line for at least four character times. If a personal computer is being used to send the download data to the MCU, there can be problems keeping characters close enough together to avoid tripping the end-of-download detect mechanism. Using $1200$ as the baud rate rather than the faster default rate may help this problem.

Asssemblers often produce S-record encoded programs which must be converted to binary before bootloading them to the MCU. The process of reading S-record data from a file and translating it to binary can be slow, depending on the personal computer and the programming language used for the translation. One strategy that can be used to overcome this problem is to translate the file into binary and store it into a RAM array before starting the download process. Data can then be read and downloaded without the translation or file-read delays.

The end-of-download mechanism goes into effect when the initial $SFF$ is received to set the baud rate. Any amount of time may pass between reset and when the $SFF$ is sent to start the download process.

EPROM/OTP versions of M68HC11 have an EPROM emulation mode — The conditions that configure the MCU for EPROM emulation mode are essentially the same as those for resetting the MCU in bootstrap mode. While $RESET$ is low and mode select pins are configured for bootstrap mode (low), the MCU is configured for EPROM emulation mode.

The port pins that are used for EPROM data I/O lines may be inputs or outputs, depending on the pin that is emulating the EPROM output enable pin ($OE$). To make these data pins appear as high-impedance inputs as they would on a non-EPROM part in reset, connect the PB7/$OE$ pin to a pull-up resistor.

Bootloading a program to perform a ROM checksum — The bootloader ROM must be turned off before performing the checksum program. To remove the boot ROM from the memory map, clear the RBOOT bit in the HPRIO register. This is normally a write-protected bit that is zero, but in bootstrap mode it is reset to one and can be written. If the boot ROM is not disabled, the checksum routine will read the contents of the boot ROM rather than the user's mask ROM or EPROM at the same addresses.

Inherent delays caused by double buffering of SCI data — This problem is troublesome in cases where one MCU is bootloading to another MCU.

Because of transmitter double buffering, there may be one character in the serial shifter as a new character is written into the transmit data register. In cases such as downloading in which this two-character pipeline is kept full, a two-character time delay occurs between when a character is written to the transmit data register and when that character finishes transmitting. A little more than one more character time delay occurs between the target MCU receiving the character and echoing it back. If the master MCU waits for the echo of each downloaded character before sending the next one, the download process takes about twice as long as it would if transmission is treated as a separate process or if verify data is ignored.

BOOT ROM VARIATIONS

Different versions of the M68HC11 have different versions of the bootstrap ROM program. Table 2 summarizes the features of the boot ROMs in 16 members of the M68HC11 Family.

The boot ROMs for the MC68HC11F1, the MC68HC711K4, and the MC68HC11K4 allow additional choices of baud rates for bootloader communications. For the three new baud rates, the first character used to determine the baud rate is not $SFF$ as it was in earlier M68HC11s. The intercharacter delay that terminates the variable-length download is also different for these new baud rates. Table 3 shows the synchronization characters, delay times, and baud rates as they relate to E-clock frequency.

COMMENTED BOOT ROM LISTINGS

Listings 3-8 contain complete commented listings of the boot ROM programs in six specific versions of the M68HC11. Other versions can be found in appendix B of the M68HC11RM/AD, M68HC11 Reference Manual.

Table 3. Bootloader Baud Rates

<table>
<thead>
<tr>
<th>Sync Character</th>
<th>Timeout Delay</th>
<th>2 MHz</th>
<th>2.1 MHz</th>
<th>3 MHz</th>
<th>3.15 MHz</th>
<th>4 MHz</th>
<th>4.2 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SFF$</td>
<td>4 Characters</td>
<td>7812</td>
<td>8192</td>
<td>11,718</td>
<td>12,288</td>
<td>15,624</td>
<td>16,838</td>
</tr>
<tr>
<td>$SFF$</td>
<td>4 Characters</td>
<td>1200</td>
<td>1260</td>
<td>1800</td>
<td>1980</td>
<td>2400</td>
<td>2520</td>
</tr>
<tr>
<td>$SFD$</td>
<td>4.9 Characters</td>
<td>9600</td>
<td>10,080</td>
<td>14,400</td>
<td>15,120</td>
<td>19,200</td>
<td>20,160</td>
</tr>
<tr>
<td>$SFD$</td>
<td>17.3 Characters</td>
<td>5208</td>
<td>5461</td>
<td>7812</td>
<td>8192</td>
<td>10,416</td>
<td>10,922</td>
</tr>
<tr>
<td>$SFD$</td>
<td>13 Characters</td>
<td>3906</td>
<td>4096</td>
<td>5859</td>
<td>6144</td>
<td>7812</td>
<td>8192</td>
</tr>
</tbody>
</table>
Listing 3. MC68HC711E9 Bootloader ROM

* BOOTLOADER Firmware for 68HC711E9 - 21 Aug 89

* Features of this bootloader are...

* Auto baud select between 7812.5 and 1200 (8 MHz)
* 0 - 512 byte variable length download
* Jump to EEPROM at $B600 if 1st download byte = $00
* PROGRAM - Utility subroutine to program EPROM
* UPLOAD - Utility subroutine to dump memory to host
* Mask I.D. at $BFD4 = $71E9

* Revision A -

* Fixed bug in PROGRAM routine where the first byte
  programmed into the EPROM was not transmitted for
  verify.
* Also added to PROGRAM routine a skip of bytes
  which were already programmed to the value desired.
* This new version allows variable length download
  by quitting reception of characters when an idle
  of at least four character times occurs

* EQUATES FOR USE WITH INDEX OFFSET = $1000

** PORTD EQU $08
** TCNT EQU $0E
** TOCI EQU $16
** TFLG1 EQU $23
** OCIF EQU $80
** SPCR EQU $28 (FOR DWOM BIT)
** BAUD EQU $2B
** SCCR2 EQU $20
** SCSR EQU $2E
** SCDAT EQU $2F
** PPROG EQU $3B
** ELAT EQU $20
** EPOG EQU $01

** MEMORY CONFIGURATION EQUATES
**
** EB600 EQU $B600 Start of EEPROM
** EB7FF EQU $B7FF End of EEPROM
** D000 EQU $D000 Start of EPROM
** FFFF EQU $FFFF End of EPROM
** RAMSTR EQU $0000
** RAMEND EQU $01FF

** DELAY CONSTANTS
**
** ODBO DELAYS EQU 3504 Delay at slow baud
** O21B DELAYF EQU 539 Delay at fast baud
** 1068 PROGDEL EQU 4200 2 ms programming delay

* At 2.1 MHz
Listing 3. MC68HC711E9 Bootloader

66
67 BF00
68 ORG $BF00
69
70 * Next two instructions provide a predictable place
71 * to call PROGRAM and UPLOAD even if the routines
72 * change size in future versions.
73
74 BF00 7EBF13
75 PROGRAM
76 JMP
77 PRGROUT
78 EPROM programming utility
79 UPLOAD
80 EQU *
81 Upload utility
82
83 * UPLOAD - Utility subroutine to send data from
84 * inside the MCU to the host via the SCI interface.
85 * Prior to calling UPLOAD set baud rate, turn on SCI
86 * and set Y-first address to upload.
87 * Bootloader leaves baud set, SCI enabled, and
88 * Y pointing at EPROM start ($D000) so these default
89 * values do not have to be changed typically.
90 * Consecutive locations are sent via SCI in an
91 * infinite loop. Reset stops the upload process.
92
93 BF03 CE1000
LDX $1000
Point to internal registers
94 BF06 1A600
UPLOOK
LDAA 0,Y
Read byte
95 BF09 1E380FC
BRCLE SCSR,X $80 *
Wait for TDRE
96 BF0D A72F
STAA SCDAT,X
Send it
97 BF0F 1808
INY
98 BF11 20F3
BRA UPLOOK
Next...
99
100 PROGRAM - Utility subroutine to program EPROM.
101 * Prior to calling PROGRAM set baud rate, turn on SCI
102 * set X=2ms prog delay constant, and set Y=first
103 * address to program. SP must point to RAM.
104 * Bootloader leaves baud set, SCI enabled, X=4200
105 * and Y pointing at EPROM start ($D000) so these
106 * default values don't have to be changed typically.
107 * Delay constant in X should be equivalent to 2 ms
108 * at 2.1 MHz X=4200; at 1 MHz X=2000.
109 * An external voltage source is required for EPROM
110 * programming.
111 * This routine uses 2 bytes of stack space
112 * Routine does not return. Reset to exit.
113
114 BF13 PRGROUT
EQU *
115 BF13 3C
PSHX
Save program delay constant
116 BF14 CE1000
LDX $1000
Point to internal registers
117 BF17 * Send $FF to indicate ready for program data
118
119 BF17 1F2E080FC
BRCLE SCSR,X $80 *
Wait for TDRE
120 BF1B 86FF
LDAA $FF
121 BF1D A72F
STAA SCDAT,X
122 BF1F WAIT1
EQU *
123 BF1F 1F2E20FC
BRCLE SCSR,X $20 *
Wait for RDRE
124 BF23 E62F
LDAB SCDAT,X
Get received byte
125 BF25 186100
CMPB $0,Y
See if already programmed
126 BF28 271D
BEQ DONEIT
If so, skip prog cycle
127 BF2A 8620
LDAA #ELAT
Put EPROM in prog mode
128 BF2C A73B
STAA PPROG,X
129 BF2E 18700
STAB 0,Y
Write the data
130 BF31 8621
LDAA #ELAT+EPGM
131 BF33 A73B
STAA PPROG,X
132 BF35 32
PULA
Pull delay constant
133 BF36 33
PUBL
into D-reg
134 BF37 37
PSHB
But also keep delay
135 BF38 36
PSHA
keep delay on stack
136 BF39 E30E
ADDP TCNT,X
Delay const + present TCNT
137 BF3B ED16
SUBB TOC1,X
Schedule OCl (2ms delay)
138 BF3D 8680
LDAA #OC1F
139 BF3F A723
STAA TFLG1,X
Clear any previous flag
140 BF41 1F2380FC
BRCLE TFLG1,X OC1F *
Wait for delay to expire
141 BF45 6F3B
CLR PPROG,X
Turn off prog voltage
142 BF47 DONEIT
EQU *

505
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Assembly</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>143</td>
<td>BF47</td>
<td>BF&lt;=800FC</td>
<td>BRCLR SCSR,X $80 * Wait for TDRE</td>
</tr>
<tr>
<td>144</td>
<td>BF48</td>
<td>18A600</td>
<td>LDX $0,Y Read from EPROM and...</td>
</tr>
<tr>
<td>145</td>
<td>BF4A</td>
<td>A72F</td>
<td>STA SCDAT,X Xmit for verify</td>
</tr>
<tr>
<td>146</td>
<td>BF50</td>
<td>1808</td>
<td>INY Point at next location</td>
</tr>
<tr>
<td>147</td>
<td>BF52</td>
<td>20CB</td>
<td>BRA WAIT1 Back to top for next</td>
</tr>
<tr>
<td>148</td>
<td></td>
<td></td>
<td>* Loops indefinitely as long as more data sent.</td>
</tr>
<tr>
<td>149</td>
<td></td>
<td></td>
<td>***************************************************************</td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
<td>* Main bootloader starts here</td>
</tr>
<tr>
<td>151</td>
<td></td>
<td></td>
<td>***************************************************************</td>
</tr>
<tr>
<td>152</td>
<td></td>
<td></td>
<td>* RESET vector points to here</td>
</tr>
<tr>
<td>153</td>
<td></td>
<td></td>
<td>***************************************************************</td>
</tr>
<tr>
<td>154</td>
<td></td>
<td></td>
<td>***************************************************************</td>
</tr>
<tr>
<td>155</td>
<td>BF54</td>
<td>BEGIN</td>
<td>EQU * Initialize stack ptr</td>
</tr>
<tr>
<td>156</td>
<td>BF54</td>
<td>BE01FF</td>
<td>LDS #RAMEND</td>
</tr>
<tr>
<td>157</td>
<td>BF56</td>
<td>CE1000</td>
<td>LDX #$1000 Point at internal regs</td>
</tr>
<tr>
<td>158</td>
<td>BF75</td>
<td>1C820</td>
<td>BSET SPCR,X $20 Select port D wire-OR mode</td>
</tr>
<tr>
<td>159</td>
<td>BF70</td>
<td>CCA20C</td>
<td>LOD #SA0C BAUD in A, SCCR2 in B</td>
</tr>
<tr>
<td>160</td>
<td>BF60</td>
<td>A72B</td>
<td>STAA BAUD,X</td>
</tr>
<tr>
<td>161</td>
<td>BF62</td>
<td>1F2E2FC</td>
<td>BRCLR SCSR,X $20 * Wait for RDRF</td>
</tr>
<tr>
<td>162</td>
<td>BF62</td>
<td>E72D</td>
<td>LOD #TOC1,X Set as default delay</td>
</tr>
<tr>
<td>163</td>
<td>BF64</td>
<td>CC021B</td>
<td>STAB SCCR2,X Rx and Tx Enabled</td>
</tr>
<tr>
<td>164</td>
<td>BF67</td>
<td>ED16</td>
<td>LDX #$DELAYF Delay for fast baud rate</td>
</tr>
<tr>
<td>165</td>
<td>BF66</td>
<td></td>
<td>STD TOC1,X</td>
</tr>
<tr>
<td>166</td>
<td>BF69</td>
<td>1C2D01</td>
<td>BSET SCCR2,X $01 Set send break bit</td>
</tr>
<tr>
<td>167</td>
<td>BF6C</td>
<td>1E0801FC</td>
<td>BRSET PORTD,X $01 * Wait for RXD pin to go low</td>
</tr>
<tr>
<td>168</td>
<td>BF70</td>
<td>1D2D01</td>
<td>BCLR SCCR2,X $01 Clear send break bit</td>
</tr>
<tr>
<td>169</td>
<td>BF73</td>
<td>F1E20FC</td>
<td>BRCLR SCSR,X $20 * Wait for RDRF</td>
</tr>
<tr>
<td>170</td>
<td>BF75</td>
<td>A62F</td>
<td>LDDA SCDAT,X Read data</td>
</tr>
<tr>
<td>171</td>
<td>BF79</td>
<td>2633</td>
<td>BNE NOTZERO Bypass JMP if not 0</td>
</tr>
<tr>
<td>172</td>
<td>BF7B</td>
<td>EB66CC</td>
<td>JMP #EEPROM Jump to EEPROM if it was 0</td>
</tr>
<tr>
<td>173</td>
<td>BF7E</td>
<td>CCCC6C</td>
<td>EQU *</td>
</tr>
<tr>
<td>174</td>
<td>BF7E</td>
<td>81FF</td>
<td>CMPA #SFF SFF will be seen as SFF</td>
</tr>
<tr>
<td>175</td>
<td>BF7F</td>
<td>2008</td>
<td>BEQ BAUDOK If baud was correct</td>
</tr>
<tr>
<td>176</td>
<td>BF79</td>
<td>1C2B31</td>
<td>BSET BAUD,X $33 Works because $22 -&gt; $33</td>
</tr>
<tr>
<td>177</td>
<td>BF89</td>
<td>CC0B00</td>
<td>LOD #DELAGS And switch to slower...</td>
</tr>
<tr>
<td>178</td>
<td>BF89</td>
<td>ED16</td>
<td>STD TOC1,X delay constant</td>
</tr>
<tr>
<td>179</td>
<td>BF8A</td>
<td>BB000000</td>
<td>BAUDOK EQU *</td>
</tr>
<tr>
<td>180</td>
<td>BF8A</td>
<td>18CE0000</td>
<td>LDX #RAMSTR Point at start of RAM</td>
</tr>
<tr>
<td>181</td>
<td>BF8E</td>
<td>WAIT</td>
<td>EQU *</td>
</tr>
<tr>
<td>182</td>
<td>BF8E</td>
<td>EC16</td>
<td>LOD TOC1,X Move delay constant to D</td>
</tr>
<tr>
<td>183</td>
<td>BF8F</td>
<td>EW0000</td>
<td>WTIMEO EQU</td>
</tr>
<tr>
<td>184</td>
<td>BF90</td>
<td>1E02007</td>
<td>BRSET SCSR,X $20 NEWONE Exit loop if RDRF set</td>
</tr>
<tr>
<td>185</td>
<td>BF94</td>
<td>8F</td>
<td>XGD0 Swap delay count to X</td>
</tr>
<tr>
<td>186</td>
<td>BF95</td>
<td>9F</td>
<td>DEX Decrement count</td>
</tr>
<tr>
<td>187</td>
<td>BF96</td>
<td>8F</td>
<td>XGD0 Swap back to D</td>
</tr>
<tr>
<td>188</td>
<td>BF97</td>
<td>26F7</td>
<td>BNE WTIME Loop if not timed out</td>
</tr>
<tr>
<td>189</td>
<td>BF99</td>
<td>200F</td>
<td>BRA STAR Quit download on timeout</td>
</tr>
<tr>
<td>190</td>
<td>BF9B</td>
<td>NEWM0NE</td>
<td>EQU *</td>
</tr>
<tr>
<td>191</td>
<td>BF9D</td>
<td>A62F</td>
<td>LDDA SCDAT,X Get received data</td>
</tr>
<tr>
<td>192</td>
<td>BF9D</td>
<td>BB700</td>
<td>STA S80,Y Store to next RAM location</td>
</tr>
<tr>
<td>193</td>
<td>BF9E</td>
<td>A72F</td>
<td>STA SCDAT,X Transmit it for handshake</td>
</tr>
<tr>
<td>194</td>
<td>BF9F</td>
<td>1808</td>
<td>INY Point at next RAM location</td>
</tr>
<tr>
<td>195</td>
<td>BF9F</td>
<td>188C0200</td>
<td>CBY #RAMEND+1 See if past end</td>
</tr>
<tr>
<td>196</td>
<td>BF9A</td>
<td>2604</td>
<td>BNE WAIT If not, Get another</td>
</tr>
<tr>
<td>197</td>
<td>BF9A</td>
<td>SC000000</td>
<td>BA RAMSTR ** EXIT to start of RAM **</td>
</tr>
<tr>
<td>198</td>
<td>BF9A</td>
<td>SC000000</td>
<td>STAR EQU</td>
</tr>
<tr>
<td>199</td>
<td>BF9A</td>
<td>CE1068</td>
<td>LDX #EPROM In X with programming delay</td>
</tr>
<tr>
<td>200</td>
<td>BF9D</td>
<td>180E0000</td>
<td>LDX #EPROMSTR Init Y with EPROM start addr</td>
</tr>
<tr>
<td>201</td>
<td>BF91</td>
<td>7E0000</td>
<td>JMP RAMSTR ** EXIT to start of RAM **</td>
</tr>
<tr>
<td>202</td>
<td>BF9B</td>
<td></td>
<td>***************************************************************</td>
</tr>
<tr>
<td>203</td>
<td>BF9B</td>
<td></td>
<td>* Block fill unused bytes with zeros</td>
</tr>
<tr>
<td>204</td>
<td>BF9B</td>
<td></td>
<td>***************************************************************</td>
</tr>
<tr>
<td>205</td>
<td>BF9B</td>
<td>44</td>
<td>F7  ** Boot ROM revision level in ASCII</td>
</tr>
<tr>
<td>206</td>
<td>BF9B</td>
<td></td>
<td>(ORG BF91)</td>
</tr>
<tr>
<td>207</td>
<td>BF9B</td>
<td>41</td>
<td>FCC &quot;A&quot;</td>
</tr>
</tbody>
</table>

---

**Listing 3. MC68HC711E9 Bootloader ROM**

**Sheet 3 of 4**

---

**506**
Listing 3. MC68HC711E9 Bootloader ROM

******************************************************************************
| Mask set I.D. ($0000 FOR EPROM PARTS) |
| (ORG $BF02) FDB $0000 |
******************************************************************************

******************************************************************************
| '711E9 I.D. - Can be used to determine MCU type |
| (ORG $BF04) FDB $71E9 |
******************************************************************************

******************************************************************************
| VECTORS - point to RAM for pseudo-vector JUMPs |
******************************************************************************

Symbol Table:

<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Value Def.</th>
<th>#</th>
<th>Line Number Cross Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUD</td>
<td>002B</td>
<td>00037</td>
<td>00160</td>
</tr>
<tr>
<td>BAUDOK</td>
<td>BFB8</td>
<td>00183</td>
<td>00179</td>
</tr>
<tr>
<td>BEGIN</td>
<td>BF54</td>
<td>00155</td>
<td>00250</td>
</tr>
<tr>
<td>DELAYF</td>
<td>021B</td>
<td>00061</td>
<td>00163</td>
</tr>
<tr>
<td>DELAYS</td>
<td>00B0</td>
<td>00060</td>
<td>00181</td>
</tr>
<tr>
<td>DONEIT</td>
<td>BF47</td>
<td>00142</td>
<td>00124</td>
</tr>
<tr>
<td>EEPMENDB</td>
<td>BF77</td>
<td>00050</td>
<td></td>
</tr>
<tr>
<td>EEPMSTR</td>
<td>B600</td>
<td>00049</td>
<td>00175</td>
</tr>
<tr>
<td>ELAT</td>
<td>0020</td>
<td>00043</td>
<td>00125</td>
</tr>
<tr>
<td>EPGM</td>
<td>0001</td>
<td>00044</td>
<td>00128</td>
</tr>
<tr>
<td>EEPMEND</td>
<td>FFF</td>
<td>00053</td>
<td></td>
</tr>
<tr>
<td>EPMSTR</td>
<td>D000</td>
<td>00052</td>
<td>00206</td>
</tr>
<tr>
<td>NEWONE</td>
<td>BF9B</td>
<td>00196</td>
<td>00189</td>
</tr>
<tr>
<td>NOTZERO</td>
<td>BF7E</td>
<td>00176</td>
<td>00174</td>
</tr>
<tr>
<td>OCIF</td>
<td>0080</td>
<td>00034</td>
<td>00136</td>
</tr>
<tr>
<td>PORTD</td>
<td>0008</td>
<td>00029</td>
<td>00168</td>
</tr>
<tr>
<td>PPRG</td>
<td>003B</td>
<td>00041</td>
<td>00126</td>
</tr>
<tr>
<td>PPGOUT</td>
<td>BF13</td>
<td>00110</td>
<td>00074</td>
</tr>
<tr>
<td>PROGDEL</td>
<td>106B</td>
<td>00063</td>
<td>00205</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>BF00</td>
<td>00074</td>
<td></td>
</tr>
<tr>
<td>RAMEND</td>
<td>01FF</td>
<td>00056</td>
<td>00156</td>
</tr>
<tr>
<td>RAMSTAR</td>
<td>0000</td>
<td>00055</td>
<td>00184</td>
</tr>
<tr>
<td>SCCR2</td>
<td>002D</td>
<td>00038</td>
<td>00162</td>
</tr>
<tr>
<td>SCDAT</td>
<td>002F</td>
<td>00040</td>
<td>00091</td>
</tr>
<tr>
<td>SCSR</td>
<td>002E</td>
<td>00039</td>
<td>00090</td>
</tr>
<tr>
<td>SPCR</td>
<td>0028</td>
<td>00036</td>
<td>00158</td>
</tr>
<tr>
<td>STAR</td>
<td>BF8A</td>
<td>00204</td>
<td>00194</td>
</tr>
<tr>
<td>TOC0</td>
<td>0000</td>
<td>00030</td>
<td>00134</td>
</tr>
<tr>
<td>TFLG1</td>
<td>0023</td>
<td>00032</td>
<td>00137</td>
</tr>
<tr>
<td>TOC1</td>
<td>0016</td>
<td>00031</td>
<td>00135</td>
</tr>
<tr>
<td>UPLOAD</td>
<td>BF03</td>
<td>00075</td>
<td></td>
</tr>
<tr>
<td>UUPLOAD</td>
<td>BF06</td>
<td>00089</td>
<td>00093</td>
</tr>
<tr>
<td>WAIT</td>
<td>BF8E</td>
<td>00186</td>
<td>00202</td>
</tr>
<tr>
<td>WATT1</td>
<td>BF1F</td>
<td>00120</td>
<td>00147</td>
</tr>
<tr>
<td>WLOOP</td>
<td>BF90</td>
<td>00188</td>
<td>00193</td>
</tr>
</tbody>
</table>

Errors: None

Labels: 35

Last Program Address: $BFFFF
Last Storage Address: $0000
Program Bytes: 0100 256
Storage Bytes: 0000 0

507
Listing 4. MC68HC11D3 Bootloader ROM

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 * BOO LADER FILEW ARE FOR MC68HC11D3 - 13 Apr 89
2 **********************************************************************
3 * Features of this bootloader are...
4 * Auto baud select between 7812 and 1200 (E = 2 MHz).
5 * 0 – 192 byte variable length download:
6 * perception of characters quits when an idle of at
7 * least four character times occurs.
8 * Jump to EPROM at $F000 if first download byte = 000.
9 * PROGRAM - Utility subroutine to program EPROM.
10 * UPLOAD - Utility subroutine to dump memory to host.
11 * Part I.D. at $BFD4 is $71D3.
12 **********************************************************************
13
14 *
15 * Equates (registers in direct space)
16 *
17 0008 PORTD EQU $08
18 0009 DDRD EQU $09
19 000E TCNT EQU $0E
20 0016 TOCI EQU $16
21 0023 TFLG1 EQU $23
22 *
23 0080 OC1F EQU $80
24 *
25 0028 SPCR EQU $28
26 002B BAUD EQU $2B
27 002C SCCR1 EQU $2C
28 002D SCCR2 EQU $2D
29 002E SCSR EQU $2E
30 002F SCDAT EQU $2F
31 003B PPTR EQU $3B
32 *
33 0020 LAT EQU $20
34 0001 EFGM EQU $01
35 *
36 003E TEST1 EQU $3E
37 003F CONFIG EQU $3F
38 *
39 *
40 *
41 * Memory configuration equates
42 *
43 $F000 ROMSTR EQU $F000 Start of ROM
44 $FFFF ROMEND EQU $FFFF End of ROM
45 *
46 $0040 RAMSTR EQU $0040 Start of RAM
47 $00FF RAMEND EQU $00FF End of RAM
48 *
49 * Delay constants
50 *
51 0DB0 DELAYS EQU $3504 Delay at slow baud
52 021B DELAYF EQU $539 Delay at fast baud
53 *
54 *
55 ORG $BF40
56 *
57 * Main bootloader starts here
58 **********************************************************************
59 *
60 *
61 BEG I N EQU *
62 *
63 $BF40 BEGIN EQU * Initialize stack ptr
64 $BF40 BE00FF LDD #RAMEND Select port D wire-OR mode
65 $BF46 142B0 CCA20C LDD #5A20C Baud in A, SCCR2 in B
66 $BF49 972B BAUD SCCR $0FF SCPx = /4, SCRx = /4
67 *
68 $BF4B D72D 20801FC STAB SCCR2 Rx and Tx enabled
69 $BF4D C021B OCL LDD #DELAYF Delay for fast baud rate
70 $BF50 DD16 STD TOCI Set as default delay
71 *
72 *
73 $BF52 142D01 BSET SCCR2 $01 Set send break bit
74 $BF55 120801FC BRSET PORTD $01 * Wait for RxO pin to go low
75 $BF59 152D01 BCLR SCCR2 $01 Clear send break bit
76 *
77 $BF5C 32Z20FC BRCLR SCCR $20 * Wait for RDRF
78 $BF60 962F LDAA SCDAT Read data
79 *
80 $BF62 2603 BNE NOTZERO Bypass jump if not $00
Jump to ROM if it was S00
SFF will be seen as SFF...
if baud was correct

Works because $22 -> $33
And switch to slower...
delay constant
Point to start of RAM
Move delay constant to X
Loop if not timed out
Quit download on timeout

Get received data
Store to next RAM location
Transmit it for handshake
Point to next RAM location
See if past end
If not, get another

** Exit to start of RAM **

* Block fill unused bytes with zero

* Boot ROM revision level in ASCII
* (ORG $BF01) FCB 0

* Mask set I.D. -
* (ORG $BF02) FDB $0000

* 11D3 I.D. - can be used to determine MCU type
* (ORG $BF04) FDB $11D3

* VECTORS - point to RAM for pseudo-vector JUMPs

** Sheet 2 of 3 **

Listing 4. MC68HC11D3 Bootloader ROM
<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Value</th>
<th>Def. #</th>
<th>Line Number</th>
<th>Cross Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUD</td>
<td>002B</td>
<td>00027</td>
<td>00066</td>
<td>00086</td>
</tr>
<tr>
<td>BAUDOK</td>
<td>BF73</td>
<td>00089</td>
<td>00084</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td>BF40</td>
<td>00062</td>
<td>00125</td>
<td></td>
</tr>
<tr>
<td>CONFIG</td>
<td>003F</td>
<td>00038</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDRD</td>
<td>0009</td>
<td>00019</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELAYF</td>
<td>021B</td>
<td>00052</td>
<td>00069</td>
<td></td>
</tr>
<tr>
<td>DELAYS</td>
<td>0380</td>
<td>00051</td>
<td>00087</td>
<td></td>
</tr>
<tr>
<td>EPGM</td>
<td>0001</td>
<td>00035</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAT</td>
<td>0020</td>
<td>00034</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEWONE</td>
<td>BF86</td>
<td>00103</td>
<td>00095</td>
<td></td>
</tr>
<tr>
<td>NOTZERO</td>
<td>BF67</td>
<td>00082</td>
<td>00080</td>
<td></td>
</tr>
<tr>
<td>OC1F</td>
<td>00B0</td>
<td>00024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PORTD</td>
<td>000B</td>
<td>00018</td>
<td>00074</td>
<td></td>
</tr>
<tr>
<td>PROG</td>
<td>003B</td>
<td>00032</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMEND</td>
<td>00FF</td>
<td>00047</td>
<td>00063</td>
<td>00108</td>
</tr>
<tr>
<td>RAMSTR</td>
<td>0040</td>
<td>00046</td>
<td>00090</td>
<td>00112</td>
</tr>
<tr>
<td>ROMEND</td>
<td>FFFF</td>
<td>00044</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROMSTR</td>
<td>F000</td>
<td>00043</td>
<td>00081</td>
<td></td>
</tr>
<tr>
<td>SCCR1</td>
<td>002C</td>
<td>00028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCCR2</td>
<td>002D</td>
<td>00029</td>
<td>00068</td>
<td>00073</td>
</tr>
<tr>
<td>SCDAT</td>
<td>002F</td>
<td>00031</td>
<td>00078</td>
<td>00104</td>
</tr>
<tr>
<td>SCSR</td>
<td>002E</td>
<td>00030</td>
<td>00077</td>
<td>00095</td>
</tr>
<tr>
<td>SPIR</td>
<td>0028</td>
<td>00026</td>
<td>00084</td>
<td></td>
</tr>
<tr>
<td>START</td>
<td>BF95</td>
<td>00111</td>
<td>00101</td>
<td></td>
</tr>
<tr>
<td>TCT</td>
<td>000E</td>
<td>00020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST1</td>
<td>003E</td>
<td>00037</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFLG1</td>
<td>0023</td>
<td>00022</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOC1</td>
<td>0016</td>
<td>00021</td>
<td>00070</td>
<td>00088</td>
</tr>
<tr>
<td>WAIT</td>
<td>BF77</td>
<td>00092</td>
<td>00109</td>
<td></td>
</tr>
<tr>
<td>WTLOOP</td>
<td>BF79</td>
<td>00094</td>
<td>00100</td>
<td></td>
</tr>
</tbody>
</table>

Errors: None
Labels: 30
Last Program Address: SBFFF
Last Storage Address: $5000
Program Bytes: $5000 192
Storage Bytes: $5000 0
LISTING 5. MC68HC711D3 Bootloader ROM

1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
10 11
11 12
12 13
13 14
14 15
15 16
16 17
17 18
18 19
19 20
20 21
21 22
22 23
23 24
24 25
25 26
26 27
27 28
28 29
29 30
30 31
31 32
32 33
33 34
34 35
35 36
36 37
37 38
38 39
39 40
40 41
41 42
42 43
43 44
44 45
45 46
46 47
47 48
48 49
49 50
50 51
51 52
52 53
53 54
54 55
55 56
56 57
57 58
58 59
59 60
60 61
61 62
62 63
63 64
64 65
65 66
66 67
67 68
68 69
69 70
70 71
71 72
72 73
73 74
74 75
75 76
76 77
77 78
78 79
79

* BOOTLOADER FIRMWARE FOR MC68HC711D3 - 28 Aug 90

* Features of this bootloader are...

* Auto baud select between 7812 and 1200 (E = 2 MHz).
* 0 - 192 byte variable length download:
* reception of characters quits when an idle of at least four character times occurs.
* Jump to EPROM at $F000 if first download byte = 500.
* PROGRAM - Utility subroutine to program EPROM.
* UPLOAD - Utility subroutine to dump memory to host.
* Part I.D. at $BF04 is $71D3.

* Revision B -
* Changed program delay to 2 mSec at E = 2 MHz.

* Revision A -
* Fixed bug in PROGRAM routine where the first byte programmed into the EPROM was not transmitted for verify.
* Also added to PROGRAM routine a skip of bytes which were already programmed to the value desired.

* Equates (registers in direct space)

30 0008 PORTD EQU $08
31 0009 DDRD EQU $09
32 000E TCNT EQU $0E
33 0016 TOC1 EQU $16
34 0023 TFLG1 EQU $23
35 0080 OC1F EQU $80
36 0028 SPCR EQU $28
37 002B BAUD EQU $2B
38 002C SCCR1 EQU $2C
39 002D SCCR2 EQU $2D
40 002E SCDR EQU $2E
41 002F SCBT EQU $2F
42 0038 PPROG EQU $38
43 0039 LAT EQU $20
44 0001 EFGM EQU $01
45 003E TEST1 EQU $3E
46 003F CONFIG EQU $3F
47 51
48 52
49 53
50 54
51 52
52 53
53 54
54 55
55 56
56 57
57 58
58 59
59 60
60 61
61 62
62 63
63 64
64 65
65 66
66 67
67 68
68 69
69 70
70 71
71 72
72 73
73 74
74 75
75 76
76 77
77 78
78 79
79

* Memory configuration equates

55 F000 EPROMSTR EQU $F000 Start of EPROM
56 FFFF EPRMEND EQU $FFFF End of EPROM
57 58 0040 RAMSTR EQU $0040 Start of RAM
59 00FF RAMEND EQU $00FF End of RAM
60
61
62
63 0DB0 DELAYS EQU 3504 Delay at slow baud
64 021B DELAYS EQU 539 Delay at fast baud
65
66 1068 PROGDEL EQU 4200 2 mSec programming delay
67
68
69

* Delay constants

70 BF00 ORG $BF00

71
72
73
74
75
76
77
78
79

* Next two instructions provide a predictable place to call PROGRAM and UPLOAD even if the routines change size in future versions.

77 BF00 7EBF10 PROGRAM JMP PGRSOUT EPROM programming utility
78 BF03 UPLOAD EQU * Upload utility

511
Listing 5. MC68HC711D3 Bootloader ROM

80  UPLOAD - Utility subroutine to send data from
81  * inside the MCU to the host via the SCI interface.
82  * Prior to calling UPLOAD set baud rate, turn on SCI
83  * and set Y-first address to upload.
84  * Bootloader leaves baud set, SCI enabled, and
85  * Y pointing at EPROM start ($FO00) so these default
86  * values do not have to be changed typically.
87  * Consecutive locations are sent via SCI in an
88  * infinite loop. Reset stops the upload process.
89
90  ******************************************************
91  BF03 18A600  LDA0 Y, Y  Read byte
92  BF03 18A600  BRCLR SCSR $80 *  Wait for TDRE
93  BF06 132E08FC  STAA SCRD  Send it
94  BF0C 1808  INY
95  BF0E 20F3  BRA UPL00O  Next...
96
97  ******************************************************
98  PROGRAM - Utility subroutine to program EPROM.
99  * Prior to calling PROGRAM set baud rate, turn on SCI
100  * set X=2ms prog delay constant, and set Y=first
101  * address to program. SP must point to RAM.
102  * Bootloader leaves baud set, SCI enabled, X=4200
103  * and Y pointing at EPROM start ($FO00) so these default
104  * values do not have to be changed typically.
105  * Delay constant in X should be equivalent to 2 ms
106  * at 2.1 MHz X=4200; at 1 MHz X=2000.
107  * An external voltage source is required for EPROM
108  * programming.
109  * This routine uses 2 bytes of stack space.
110  * Routine does not return. Reset to exit.
111
112  ***********************************************.****
113  BF10 132E00FC  BRCLR SCSR $80 *  Wait for TDRE
114  BF16 86FF  LDA0 #FF  Next...
115  BF18 972F  STAA SCRD
116
117  ***********************************************.****
118  BF18 132E20FC  BRCLR SCSR $20 *  Wait for RDRF
119  BF1C 6E2F  LDAB SCRD  Get received byte
120  BF1E 18E100  CMPB $0, $Y  See if already programmed
121  BF20 420D  BEQ DONEIT  If so, skip prog cycle
122  BF22 8620  LDA0 #LAT  Put EPROM in prog mode
123  BF25 973B  STAA PPFG  Delay const + present TCNT
124  BF27 187E00  STAB Y, Y  Write data
125  BF2A 8621  LDA0 #LAT+EPGM
126  BF2C 973B  STAA PPFG  Turn on prog voltage
127  BF2E 3C  PSHX  Save delay on stack
128  BF2F 9F  XGDX  Put delay into D-reg
129  BF30 38  PULLX  Save delay in X
130  BF31 D30E  ADDD TCNT
131  BF33 D616  STD TOC1  Schedule OC1 (prog delay)
132  BF35 8680  LDA0 #OClF
133  BF37 9723  STAA TFLG1  Clear any previous flag
134
135  ***********************************************.****
136  BF39 132380FC  BRCLR TFLG1 OClF *  Wait for delay to expire
137  BF3D 7F033B  CLR PPFG  Turn off prog voltage
138  BF40  DONEIT  EQU *
139
140  ***********************************************.****
141  BF40 132E00FC  BRCLR SCSR $80 *  Wait for TDRE
142  BF44 18A600  LDAA $0, Y  Read from EPROM and...
143  BF47 972F  STAA SCRD  Xmit for verify
144  BF49 1808  INY  Point to next location
145  BF4B 20CB  BRA WAIT1  Back to top for next
146
147  ***********************************************.****
148  * Loops indefinitely as long as more data sent.
149
150  ******************************************************
151  BF4D BEGIN  EQU *
152  BF4D 8E00FF  LDS #RAMEND  Initialize stack pointer
153  BF50 142820  BSET SPCHR $20  Select port D wire-OR mode
154  BF53 CC200C  LDD #A20C  Baud in A, SCCR2 in B
155  BF56 972B  STAA BAUD  SCPx = /4, SCRx = /4
156
157  ***********************************************.****
158  BF58 D72D  STAB SCCR2  Rx and Tx enabled
159  BF5A CC021B  LDD #DELAYF  Delay for fast baud rate

512
Listing 5. MC68HC711D3 Bootloader ROM

BSZ $BFD1-

** Exit to start of RAM **

* Block fill unused bytes with zero

BSZ $BFD1-

** Boot ROM revision level in ASCII

* (ORG $BFD1) "B"

******************************************************************************

* Mask set I.D. ($0000 for EPROM parts)

* (ORG $BFD2) FDB $0000

******************************************************************************

* 71D3 I.C. - can be used to determine MCU type

* (ORG $BFD4) FDB $71D3

******************************************************************************

* VECTORS - point to RAM for pseudo-vector JUMPs

******************************************************************************

* Send BREAK to signal ready for download

BSET SCCR2 $01

Set send break bit

BSET PORTD $01

Wait for RxD pin to go low

BCLR SCCR2 $01

Clear send break bit

* Data will be $00 if BREAK or $00 received

BNE NOTZERO

Bypass jump if not $00

Jump to EEPROM if $00

* Or else change to /104 (/13 & /8) 1200 @ 2MHz

BSET BAUD $33

Works because $22 -> $33

LDX DELAYS

And switch to slower delay constant

STD TOC1

delay constant

BAUDOK

Point to start of RAM

WAIT

Move delay constant to X

LDX TOC1

Point to start of RAM -*

**-**-**-**-**-**-**-**

Point to next RAM location

Store to next RAM location

Point to next RAM location

Get received data

Store to next RAM location

Transmit it for handshake

Point to next RAM location

Quit download on timeout

Exit loop if not timed out

Loop if not timed out

Init X with program delay

Init Y with EPROM start addr

** Exit to start of RAM **
Listing 6. MC68HC11F1 Bootloader ROM

1 * BOOTLOADER Firmware for MC68HC11F1 - 04 May 90
2 ******************************************************
3 * Features of this bootloader are...
4 *
5 * Auto baud select between 7812, 1200, 9600, 5208
6 * and 3906 (f = 2 MHz).
7 * 0 - 1024 byte variable length download:
8 * reception of characters quits when an idle of at
9 * least four character times occurs. (Note: at 9600
10 * baud rate this is almost five bit times and at
11 * 5208 and 3906 rates the timeout is even longer).
12 * Jump to EEPROM at SFED0 if first download byte = 00.
13 * Part I.D. at $BF04 is $81F1.
14 ******************************************************
15 * Revision B -
16 *
17 * Added new baud rates: 5208, 3906.
18 ******************************************************
19 * Revision A -
20 *
21 * Added new baud rate: 9600.
22 ******************************************************
23 *
24 * Equates (use with index offset = $1000)
25 *
27 0008 PORTD EQU $08
28 0009 DDRO EQU $09
29 0016 TOC1 EQU $16
30 0028 SCRC EQU $28
31 002B BAUD EQU $2B
32 002C SCRC1 EQU $2C
33 002D SCRC2 EQU $2D
34 002E SCST EQU $2E
35 002F SCST1 EQU $2F
36 002B SPROG EQU $3B
37 003E TEST1 EQU $3E
38 003F CONFIG EQU $3F
39 *
40 * Memory configuration equates
41 *
42 FEED EEPSTR EQU $FE00 Start of EEPROM
43 FFED EEPEnd EQU $FFFF End of EEPROM
44 *
45 0000 RAMSTR EQU $0000 Start of RAM
46 03FF RAMEND EQU $03FF End of RAM
47 *
48 * Delay constants
49 *
50 0DB0 DELAYS EQU 3504 Delay at slow baud rate
51 021B DELAYF EQU 539 Delay at fast baud rates
52 *
53 ******************************************************
54 $BF00 ORG $BF00
55 ******************************************************
56 * Main bootloader starts here
57 ******************************************************
58 *
59 $BF00 BEGIN EQU *
60 $BF00 $E03FF LDX #DATA EQU X points to registers
61 $BF00 $E1000 BSET SPCR X $20 Select port D wire-OR mode
62 $BF00 $C2820 LDD #8000 BAUD X $28
63 $BF00 $EBO00 BSET SCRC X $2B
64 $BF00 $A72B LDD #8000 BAUD X SCRC = /13 SCRC = /1
65 *
66 $BF00 $E72D STAB CURRENT $SCRC2 X Rx and Tx enabled
67 $BF00 $CC01B LDD #DELAYF Delay for fast baud rates
68 $BF00 $ED16 STD TOC1 X Set as default delay
69 *
70 * Send BREAK to signal start of download
71 $BF15 $12D01 BSET SCRC2 X $01 Set send break bit
72 $BF18 $E801FC BRST PORTD X $01 * Wait for RxD pin to go low
73 $BF1C $D2D01 BCLR SCRC2 X $01 Clear send break bit
74 *
75 $BF1F $F2E00FC BRCLR SCST $0200 * Wait for RDRF
76 $BF23 $A62F LDAA SCDAT X Read data
77 *
515
Listing 6. MC68HC11F1 Bootloader ROM

BNE NOTZERO |
Bypass jump if not $00

JMP EEPSFR |
Jump to EEPROM if it was $00

NOTZERO EQU * |

* Check div by 13 (6000 baud at 2 MHz) |
CMPA #$F0 |
$0 will be seen as $00...

BEQ BAUDOK |
if baud was correct

* Check div by 104 (1200 baud at 2 MHz) |
LDAB #$33 |
Initialize B for this rate

$FF will be seen as $80...

if baud was correct

* Check div by 32 (9600 baud at 2 MHz) |
* (equals: 8192 baud at 4.2 MHz) |
Initialize B for this rate

$FD shows as bit 5 clear... |
if baud was correct

* Change div by 16 (7680 baud at 2 MHz) |
* (equals: 8192 baud at 2.1 MHz) |
Initialize B for this rate

* Change div by 24 (5080 baud at 2 MHz) |
* (equals: 8192 BAUD at 3.15 MHz) |
By default

SLOBAUD EQU * |
Store baud rate

STAB BAUD,X |
Switch to slower...

delay constant

EQU * |

LDY #RAMSTR |
Point to start of RAM

WAIT EQU * |
Move delay constant to D

LDD TOC1,X |
Exit loop if RDRF set

EQU * |

BGST scsr,X $20 NEWONE |
Swap delay count to X

Swap delay count to X |
Decrement count

BAUDOK |
Swap back to D

GXX |
Loop if not timed out

** Exit to start of RAM ** |

BRA STAR |
Quit download on timeout

NEWONE EQU * |
Get received data

LDAA scdat,X |
Store to next RAM location

STAA $50,Y |
Transmit it for handshake

STAA scdat,X |
Point to next RAM location

CPY $RAMEND+1 |
See if past end

BNE WAIT |
If not, get another

STAR EQU * |
** Exit to start of RAM **

JMP $0000 |

* Block fill unused bytes with zero

BSZ $BFDF--* |

* Boot ROM revision level in ASCII
*(ORG $BFDF)

ECC "B" |

* Mask set I.D. - ($0000 for ROMless parts)
*(ORG $BFDF)

FDB $0000 |

* 11FL I.D. - can be used to determine MCU type
*(ORG $BFDF)

FDB $0F1 |

*****************************************************
Listing 6. MC68HC11F1 Bootloader ROM

148
149
150 BFD6 00C4  FDB $100-60  SCI
151 BFD8 00C7  FDB $100-57  SPI
152 BFD9 00CA  FDB $100-54  PULSE ACCUM INPUT EDGE
153 BFD0 00CD  FDB $100-51  PULSE ACCUM OVERFLOW
154 BFE0 00D0  FDB $100-48  TIMER OVERFLOW
155 BFE2 00D6  FDB $100-42  TIMER OUTPUT COMPARE 5
156 BFE4 00D9  FDB $100-39  TIMER OUTPUT COMPARE 4
158 BFE6 00DC  FDB $100-36  TIMER OUTPUT COMPARE 3
159 BFE8 00DF  FDB $100-33  TIMER OUTPUT COMPARE 2
160 BFEA 00E2  FDB $100-30  TIMER INPUT CAPTURE 3
161 BFEC 00E5  FDB $100-27  TIMER INPUT CAPTURE 2
162 BFEE 00E8  FDB $100-24  TIMER INPUT CAPTURE 1
163 BFF0 00EB  FDB $100-21  REAL TIME INT
164 BFF2 00EE  FDB $100-18  IRQ
165 BFF4 00F1  FDB $100-15  XIRQ
166 BFF6 00F4  FDB $100-12  SWI
167 BFF8 00F7  FDB $100-9  ILLEGAL OP-CODE
168 BFFA 00FA  FDB $100-6  COP FAIL
169 BFFC 00FD  FDB $100-3  CLOCK MONITOR
170 BFTE BF00  FDB BEGIN  RESET
171 C000  END

Symbol Table:

Symbol Name  Value  Def.#  Line Number  Cross Reference
BAUD        002B  *00031  00064 00096 00104
BAUDOK      BF4B  *00107  00083 00098
BEGIN       BF0D  *00059  00170
CONFIG      003F  *00038
DORD        0009  *00028
DELAYF      0218  *00051  00067
DELAYS      0080  *00050  00105
EEPEND      FFFF  *00043
EEPSTR      FE00  *00042  00079
ERRORS      BF5C  *00120  00113
NOTZERO     BF2A  *00080  00078
PORTD       0008  *00027  00072
PPREG       0038  *00036
RAMEND      03F8  *00046  00060 00125
RAMSTR      0000  *00045  00108 00129
SCCR1       002C  *00032
SCCR2       002D  *00033  00066 00071 00073
SCDAT       002F  *00035  00076 00121 00123
SCSR        002E  *00034  00075 00113
SIOBAUD     BF44  *00103  00087 00092
SPCR        0028  *00030  00062
STAR        BF6B  *00128  00118
TEST1       003E  *00037
TSCI        0016  *00029  00068 00106 00111
WAIT        BF4F  *00110  00126
WTLOOP       BF51  *00112  00117

Errors: None
Labels: 26
Last Program Address: $BFFF
Last Storage Address: $0000
Program Bytes: $100 256
Storage Bytes: $0000 0
Listing 7. MC68HC11K4 Bootloader ROM

1
2 * BOOTLOADER Firmware FOR MC68HC11K4 - 18 Jul 90
3 *******************************************
4 * Features of this bootloader are...
5 *
6 * Auto baud select between 7812, 1200, 9600, 5208
7 * and 3906 (E = 2 MHz).
8 * 0 - 768 byte variable length download:
9 * reception of characters quits when an idle of at
10 * least four character times occurs. (Note: at 9600
11 * baud rate this is almost five bit times and at
12 * 5208 and 3906 rates the timeout is even longer).
13 * Jump to EEPROM at $0D80 if first download byte = $00.
14 * PROGRAM - Utility subroutine to program EPROM.
15 * UPLOAD - Utility subroutine to dump memory to host.
16 * Part I.D. at $2BD4 is $044B.
17 *******************************************
18 *
19 * Equates (registers in direct space)
20 *
21 21 0004 PORTB EQU $04
22 22 0005 PORTF EQU $05
23 23 0008 PORTD EQU $08
24 24 0009 DDRD EQU $09
25 *
26 26 000E TCNT EQU $0E
27 27 0016 TC1 EQU $16
28 28 0023 TFLG1 EQU $23
29 *
30 0080 OCIF EQU $80
31 *
32 002B EPROG EQU $2B
33 *
34 0020 ELAT EQU $20
35 0001 EPGM EQU $01
36 *
37 003B PPROM EQU $3B
38 003E TEST1 EQU $3E
39 003F CONFIG EQU $3F
40 *
41 0070 SCBD EQU $70
42 0072 SCCR1 EQU $72
43 0073 SCCR2 EQU $73
44 0074 SCBR1 EQU $74
45 0075 SCBR2 EQU $75
46 0076 SCDRH EQU $76
47 0077 SCDRL EQU $77
48 *
49 * Memory configuration equates
50 *
51 0D80 EEPMSTR EQU $0D80 Start of EEPROM
52 0FF FEEPMEND EQU $0FF End of EEPROM
53 *
54 2000 ROMSTR EQU $2000 Start of ROM
55 7FF ROMEND EQU $7FF End of ROM
56 *
57 0080 RAMSTR EQU $0080 Start of RAM
58 037F RAMEND EQU $037F End of RAM
59 *
60 * Delay constants
61 *
62 15AB DELAYS EQU $5547 Delay at slow baud rate
63 0356 DELAYF EQU $854 Delay at fast baud rates
64 *
65 1068 PROGDEL EQU $4200 2 mSec programming delay
66 *
67 *
68 BE40 CYCLCOD EQU $BE40 EPROM cycling code (TEST)
69 *
70 *******************************************
71 *
72 * Main bootloader starts here
73 *******************************************
74 *
75 * RESET vector points to here
76 *
77 BF00 BEGIN EQU * Initialize stack ptr
78 BF00 8E037F LDS #RAMEND
79 *
80 * Special jump for EPROM Cycling routine
81 * (This is intended for factory test only)
* If ports B and F both have \$1001 0110 on them ...
LDD \#9696
CPD PORTB
BNE CONTINU

* then execute the cycling code
JMP CYCLODD
CONTINU EQU *

LDD \#0001A
STD SCBD
LDD \#6400C
STD SCCR1
LDAB \#DELAYF
STD TOCI
Set as default delay

* Send BREAK to signal ready for download
BSET SCCR2 \$01
BRSET PORTD \$01
Wait for RXD pin to go low
BCLR SCCR2 \$01
Clear send break bit

* Data will be \$00 if BREAK or \$00 received
LDAB \#00
BITA \#08
BNE BAUDOK

Check div by 26 (9600 baud at 2 MHz)
CMFA \#F0
BEQ BAUOOK

Check div by 208 (1200 baud at 2 MHz)
LDAB \#SD0
CMFA \#80
if baud was correct
BNE SLOBAUD

Check div by 64 (3906 baud at 2 MHz)
LDAB \#40
CMFA \#80
if baud was correct
BNE SLOBAUD

Change to div by 32 (7812 baud at 2 MHz)
LDAB \#$20
STAB SCDRL+1
BITA \#$08
BNE BAUDOK

Change to div by 48 (5208 baud at 2 MHz)
LDAB \#$30
STAB #RAMEND+1

About to start at \$FF shows as bit 3 set ...
LDAB \#$FF

About to start at \$FD shows as bit 5 clear ...
LDAB \#$FF

By default
LDAB \#80

Store baudrate
STAB \#SCBD+1

Switch to slower ...
LDD \#DELAIS

delay constant
STD TOCI

Point to start of RAM
LDY \#RAMSTR

Move delay constant to X
LDX TOCI

Exit loop if RDRF set
BRSET SCSR1 \$20 NEWONE
Decrement count
DEX

Loop if not timed out
BNE WTLOOP

Quit download on timeout
BRA STAR

Get received data
LDAA \#SCDRL

Store to next RAM location
STAA \#SD0,Y

Transmit it for handshake
STAA SCDRL

Point to next RAM location
INY

See if past end
CPY \#RAMEND+1

If not, get another
BNE WTLOOP

** Exit to start of RAM **

* Block fill unused bytes with zero

---

519
Listing 7. MC68HC11K4 Bootloader ROM

156 BF74 000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
000000000000
BSZ $BFD1-*

157
158
159
160
161 BFD1 30
162
163
164
165 BFD2 0000
166
167
168
169
170 BFD4 044B
171 172
173
174 BF66 00C4
175 BF60 00C7
176 BF6A 00CA
177 BF6D 00CD
178 BF6E 00DC
179 BF6F 00DD
180 BF62 00D6
181 BF64 00D9
182 BF66 00DC
183 BF68 00DF
184 BF6A 00E2
185 BF6C 00E5
186 BF6E 00E8
187 BF6F 00EB
188 BF70 00EE
189 BF74 00F1
190 BF76 00F4
191 BF78 00F7
192 BF7A 00FA
193 BF7C 00FD
194 BF7E 0000
195 C000

BSZ $BFD1-*

* Boot ROM revision level in ASCII
* (ORG $BFD1)
FCC "0"

* Mask set I.D. - set with user's ROM code mask layer
* (ORG $BFD2)
FDB $0000 Reserve 2 bytes

* 1K4 I.D. - can be used to determine MCU type
* (note: $4B = K in ASCII)
* (ORG $BFD4)
FDB $044B

* VECTORS - point to RAM for pseudo-vector JUMPs

174 BF66 00C4
175 BF60 00C7
176 BF6A 00CA
177 BF6D 00CD
178 BF6E 00DC
179 BF6F 00DD
180 BF62 00D6
181 BF64 00D9
182 BF66 00DC
183 BF68 00DF
184 BF6A 00E2
185 BF6C 00E5
186 BF6E 00E8
187 BF6F 00EB
188 BF70 00EE
189 BF74 00F1
190 BF76 00F4
191 BF78 00F7
192 BF7A 00FA
193 BF7C 00FD
194 BF7E 0000
195 C000

SC1
SPI
PULSE ACCUM INPUT EDGE
PULSE ACCUM OVERFLOW
TIMER OVERFLOW
TIMER OUTPUT COMPARE 5
TIMER OUTPUT COMPARE 4
TIMER OUTPUT COMPARE 3
TIMER OUTPUT COMPARE 2
TIMER OUTPUT COMPARE 1
TIMER INPUT CAPTURE 3
TIMER INPUT CAPTURE 2
TIMER INPUT CAPTURE 1
REAL TIME INT
IRQ
XIRQ
SWI
ILLEGAL OP-CODE
COP FAIL
CLOCK MONITOR
CLOCK MONITOR
RESET
### Symbol Table:

<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Value</th>
<th>Def.#</th>
<th>Line Number</th>
<th>Cross Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUDOK</td>
<td>BF53</td>
<td>*00132</td>
<td>00108</td>
<td>00123</td>
</tr>
<tr>
<td>BEGIN</td>
<td>BF00</td>
<td>*00076</td>
<td>00194</td>
<td></td>
</tr>
<tr>
<td>CONFIG</td>
<td>003F</td>
<td>*00039</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONTINU</td>
<td>BFGE</td>
<td>*00087</td>
<td>00084</td>
<td></td>
</tr>
<tr>
<td>CYCLCOD</td>
<td>BK40</td>
<td>*00068</td>
<td>00086</td>
<td></td>
</tr>
<tr>
<td>DDRO</td>
<td>0009</td>
<td>*00024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELAYF</td>
<td>0356</td>
<td>*00063</td>
<td>00093</td>
<td></td>
</tr>
<tr>
<td>DELAYS</td>
<td>15AB</td>
<td>*00062</td>
<td>00130</td>
<td></td>
</tr>
<tr>
<td>EEPMEND</td>
<td>0FFF</td>
<td>*00052</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEPMSTR</td>
<td>0D80</td>
<td>*00051</td>
<td>00104</td>
<td></td>
</tr>
<tr>
<td>ELAT</td>
<td>0020</td>
<td>*00034</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPGM</td>
<td>0001</td>
<td>*00035</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPROG</td>
<td>002B</td>
<td>*00032</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEWNONE</td>
<td>BF62</td>
<td>*00143</td>
<td>00138</td>
<td></td>
</tr>
<tr>
<td>NOTIEPRO</td>
<td>BF32</td>
<td>*00105</td>
<td>00133</td>
<td></td>
</tr>
<tr>
<td>OCIF</td>
<td>0080</td>
<td>*00030</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PORTB</td>
<td>0004</td>
<td>*00021</td>
<td>00083</td>
<td></td>
</tr>
<tr>
<td>PORTD</td>
<td>0008</td>
<td>*00023</td>
<td>00097</td>
<td></td>
</tr>
<tr>
<td>PORTF</td>
<td>0005</td>
<td>*00022</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPROM</td>
<td>003B</td>
<td>*00037</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGDEL</td>
<td>1068</td>
<td>*00065</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMEND</td>
<td>037F</td>
<td>*00058</td>
<td>0077</td>
<td>00148</td>
</tr>
<tr>
<td>RAMSTR</td>
<td>0080</td>
<td>*00057</td>
<td>00133</td>
<td>00152</td>
</tr>
<tr>
<td>ROMEND</td>
<td>07FF</td>
<td>*00055</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROMSTR</td>
<td>2000</td>
<td>*00054</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCBD</td>
<td>0070</td>
<td>*00041</td>
<td>0090</td>
<td>00121</td>
</tr>
<tr>
<td>SCCR1</td>
<td>0072</td>
<td>*00042</td>
<td>0092</td>
<td></td>
</tr>
<tr>
<td>SCCR2</td>
<td>0073</td>
<td>*00043</td>
<td>0096</td>
<td>00098</td>
</tr>
<tr>
<td>SCDRH</td>
<td>0076</td>
<td>*00046</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCDRL</td>
<td>0077</td>
<td>*00047</td>
<td>0101</td>
<td>00144</td>
</tr>
<tr>
<td>SCFSR1</td>
<td>0074</td>
<td>*00044</td>
<td>0100</td>
<td>00138</td>
</tr>
<tr>
<td>SCFSR2</td>
<td>0075</td>
<td>*00045</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOGAUD</td>
<td>BF4C</td>
<td>*00128</td>
<td>00112</td>
<td>00117</td>
</tr>
<tr>
<td>STAR</td>
<td>BF71</td>
<td>*00151</td>
<td>00141</td>
<td></td>
</tr>
<tr>
<td>TCNT</td>
<td>000E</td>
<td>*00026</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST1</td>
<td>003E</td>
<td>*00038</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFLAG</td>
<td>0023</td>
<td>*00028</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOC1</td>
<td>0016</td>
<td>*00027</td>
<td>0094</td>
<td>00131</td>
</tr>
<tr>
<td>WAIT</td>
<td>BF57</td>
<td>*00135</td>
<td>00149</td>
<td></td>
</tr>
<tr>
<td>WTLOOP</td>
<td>BF59</td>
<td>*00137</td>
<td>00140</td>
<td></td>
</tr>
</tbody>
</table>

#### Errors: None

#### Labels: 40

- Last Program Address: $BFFF
- Last Storage Address: $0000
- Program Bytes: $0100 256
- Storage Bytes: $0000 0

---

521
Listing 8. MC68HC711K4 Bootloader ROM

0004 30 0005 31 0008 32 0023 33 0080 34 0074 35 0076 36 0077 37 0078 38 0079 39 0080 40 002B 41 0020 42 0001 43 003B 44 003E 45 0080 46 007F 47 0074 48 0075 49 0076 50 0077 51 0078 52 0079 53 007A 54 007B 55 007C 56 007D 57 007E 58 007F 59 0D80 60 0FFF 61 7FFF 62 0080 63 037F 64 15AB 65 0356 66 1068 67 7E40 68 BF00

******************************************************

---

Copyright © 1990

---

* BOOTLOADER Firmware for MC68HC711K4 - 25 Apr 90

* Features of this bootloader are...

* Auto baud select between 7812, 1200, 9600, 5208
* and 3906 (E = 2 MHz).
* 0 - 768 byte variable length download.
* Reception of characters quits when an idle of at least four character times occurs. (Note: at 9600 baud rate this is almost five bit times and at 5208 and 3906 rates the timeout is even longer).
* Jump to EEPROM at $D080 if first download byte = $00.

* PROGRAM - Utility subroutine to program EEPROM.

* UPLOAD - Utility subroutine to dump memory to host.

* Part I.D. at $BF04 is $744B.

* Revision B -

* Added new baud rates: 5208, 3906.

* Revision A -

* Added new baud rate: 9600.

* Equates (registers in direct space)

* PORTB EQU $04
* PORTF EQU $05
* PORTD EQU $08
* DDRD EQU $09
* TCNT EQU $0E
* TOC1 EQU $16
* TFLG1 EQU $23
* OC1F EQU $80
* EPROG EQU $2B
* ELAT EQU $20
* EPGM EQU $01
* PFROG EQU $3B
* TEST1 EQU $3E
* CONFIG EQU $3F
* SCBD EQU $70
* SCCR1 EQU $72
* SCCR2 EQU $73
* SCSR1 EQU $74
* SCSR2 EQU $75
* SCDRH EQU $76
* SCDRL EQU $77
* EEMSTR EQU $D080
* EEMEND EQU $0FFF
* EPMSTR EQU $2000
* EPMEND EQU $7FFF
* RAMSTR EQU $0080
* RAMEND EQU $037F
* DELAYS EQU $5547
* DELAYF EQU $854
* PROGDEL EQU $4200
* CYCLCOD EQU $BE40
* ORG $BF00
Listing 8. MC68HC711K4 Bootloader ROM

82  * Next two instructions provide a predictable place
83  * to call PROGRAM and UPLOAD even if the routines
84  * change size in future versions.
85  *
86 BF00 7E8F1D
87 BF03  UPLOAD  EQU  *  Upload utility
88
89 ******************************************************
90  * UPLOAD - Utility subroutine to send data from
91  * inside the MCU to the host via the SCI interface.
92  * Prior to calling UPLOAD set baud rate, turn on SCI
93  * and set Y=first address to upload.
94  * Bootloader leaves baud set, SCI enabled.
95  * Consecutive locations are sent via SCI in an
96  * infinite loop. Reset stops the upload process.
97  ******************************************************
98 BV03 8D0D
99  BSR  INIT
100  Initialization subroutine
101
102 BF05 18A600
103 BF08 137480FC
104 BF0C 9777
105 BF0E 188F
106 BF10 20F3
107
108 ******************************************************
109  * Initialization subroutine - Forces EPROM to be
110  * enabled at $2000 so it is not overlapped by the
111  * BOOTLOADER firmware.
112  * User's address in index Y is adjusted to point to
113  * EPROM in this space instead of $A000.
114 ******************************************************
115 BF12 860F
116 BF14 973F
117 BF16 188F
118 BF18 847F
119 BF1A 18FF
120 BF1C 39
121
122 ******************************************************
123  * PROGRAM - Utility subroutine to program EPROM.
124  * Prior to calling PROGRAM set baud rate, turn on SCI
125  * set X=2ms prog delay constant, and set Y=first
126  * address to program. SP must point to RAM.
127  * Bootloader leaves baud set, and SCI enabled so these
128  * default values do not have to be changed typically.
129  * Delay constant in X should be equivalent to 2 ms
130  * at 2.1 MHz X=4200; at 1 MHz X=2000, at 4MHz X=8000.
131  * An external voltage source is required for EPROM
132  * programming.
133  * This routine uses 2 bytes of stack space.
134  * Routine does not return. Reset to exit.
135 ******************************************************
136 BF1D  PGRROUT
137 BF1D 8DF3
138
139 BF1F 137480FC
140 BF23 86FF
141 BF25 9777
142
143 ******************************************************
144 BF27  WAIT1  EQU  *
145
146 BF2B 137420FC
147 BF2D 188100
148 BF30 271D
149 BF32 8620
150 BF34 972B
151 BF36 18E700
152 BF39 8621
153 BF3B 972B
154 BF3D 3C
155 BF3E 32
156 BF40 D30E
157 BF42 DD16
158 BF44 8680
159 BF46 9723
160 BF48 132380FC
161
*
523
Listing 8. MC68HC711K4 Bootloader ROM

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>162</td>
<td>BF4C 7F002B</td>
<td>CLR EPROG</td>
</tr>
<tr>
<td>163</td>
<td>BF4F</td>
<td>DONEIT EQU *</td>
</tr>
<tr>
<td>164</td>
<td>BF4F 137480FC</td>
<td>BRCRL SCSR1 $80 *</td>
</tr>
<tr>
<td>165</td>
<td>BF53 18A600</td>
<td>LDAA $0,Y</td>
</tr>
<tr>
<td>166</td>
<td>BF56 9777</td>
<td>STA SCDDL</td>
</tr>
<tr>
<td>167</td>
<td>BF58 1808</td>
<td>INY</td>
</tr>
<tr>
<td>168</td>
<td>BF5A 20CB</td>
<td>BRA WAIT</td>
</tr>
<tr>
<td>169</td>
<td></td>
<td>* Loops indefinitely as long as more data sent.</td>
</tr>
<tr>
<td>170</td>
<td></td>
<td>*******************************************************</td>
</tr>
<tr>
<td>171</td>
<td></td>
<td>* Main bootloader starts here</td>
</tr>
<tr>
<td>172</td>
<td></td>
<td>*******************************************************</td>
</tr>
<tr>
<td>173</td>
<td>BF5C 8E037F</td>
<td>EQU</td>
</tr>
<tr>
<td>174</td>
<td>BF5C</td>
<td>INITIALIZE stack ptr</td>
</tr>
<tr>
<td>175</td>
<td>BF5F CC9696</td>
<td>LDD #$9696</td>
</tr>
<tr>
<td>176</td>
<td>BF62 1A9304</td>
<td>CPD PORTR</td>
</tr>
<tr>
<td>177</td>
<td>BF65 2603</td>
<td>BNE CONTINU</td>
</tr>
<tr>
<td>178</td>
<td></td>
<td>... then execute the cycling code</td>
</tr>
<tr>
<td>179</td>
<td>BF67 7EEBE40</td>
<td>JMP CYCLCOD</td>
</tr>
<tr>
<td>180</td>
<td>BF6A</td>
<td>CONTINU EQU *</td>
</tr>
<tr>
<td>181</td>
<td>BF6A</td>
<td>Initialize baud for...</td>
</tr>
<tr>
<td>182</td>
<td>BF6A CC001A</td>
<td>STD SCBD</td>
</tr>
<tr>
<td>183</td>
<td>BF6K 2400C</td>
<td>LDD #$400C</td>
</tr>
<tr>
<td>184</td>
<td>BF72 2D72</td>
<td>STD SCCR1</td>
</tr>
<tr>
<td>185</td>
<td>BF74 CC0356</td>
<td>LDD #$8000</td>
</tr>
<tr>
<td>186</td>
<td>BF77 DD16</td>
<td>STD TOC1</td>
</tr>
<tr>
<td>187</td>
<td></td>
<td>* Send BREAK to signal ready for download</td>
</tr>
<tr>
<td>188</td>
<td>BF79 147301</td>
<td>BSET SCCR2 $01</td>
</tr>
<tr>
<td>189</td>
<td>BF8C 120801FC</td>
<td>BSET PORTD $01 *</td>
</tr>
<tr>
<td>190</td>
<td>BF80 157301</td>
<td>BCLR SCCR2 $01</td>
</tr>
<tr>
<td>191</td>
<td>BF81 2603</td>
<td>BRCRL SCSR1 $20 *</td>
</tr>
<tr>
<td>192</td>
<td>BF8B 7EEB00</td>
<td>LDAA SCDDL</td>
</tr>
<tr>
<td>193</td>
<td>BF8B</td>
<td>Read data</td>
</tr>
<tr>
<td>194</td>
<td>BF8F 0000H</td>
<td>JMP EEPMSTR</td>
</tr>
<tr>
<td>195</td>
<td>BF8F</td>
<td>NOTZERO EQU</td>
</tr>
<tr>
<td>196</td>
<td>BF8F</td>
<td>* Check div by $3000 (6000 baud at 2 MHz)</td>
</tr>
<tr>
<td>197</td>
<td>BF8F</td>
<td>CMFA #$F0</td>
</tr>
<tr>
<td>198</td>
<td>BF90 271D</td>
<td>BEQ BAUDOK</td>
</tr>
<tr>
<td>199</td>
<td>BF92 C600</td>
<td>* Check div by 208 (1200 baud at 2 MHz)</td>
</tr>
<tr>
<td>200</td>
<td>BF94 1808</td>
<td>LDAB #$D0</td>
</tr>
<tr>
<td>201</td>
<td>BF94</td>
<td>CMFA #$80</td>
</tr>
<tr>
<td>202</td>
<td>BF96 271D</td>
<td>BEQ SLOBAUD</td>
</tr>
<tr>
<td>203</td>
<td>BF98 2100</td>
<td>* Check div by 64 (3906 baud at 2 MHz)</td>
</tr>
<tr>
<td>204</td>
<td>BF98</td>
<td>(equals: 8192 baud at 4.2 MHz)</td>
</tr>
<tr>
<td>205</td>
<td>BF9A 8520</td>
<td>LDAB #$40</td>
</tr>
<tr>
<td>206</td>
<td>BF9A</td>
<td>BITA #$20</td>
</tr>
<tr>
<td>207</td>
<td>BF9C 207A</td>
<td>BEQ SLOBAUD</td>
</tr>
<tr>
<td>208</td>
<td>BF9C</td>
<td>* Change to div by 32 (7812 baud at 2 MHz)</td>
</tr>
<tr>
<td>209</td>
<td>BF9E C620</td>
<td>(equals: 8192 baud at 2.1 MHz)</td>
</tr>
<tr>
<td>210</td>
<td>BF9E</td>
<td>LDAB #$20</td>
</tr>
<tr>
<td>211</td>
<td>BF9F D771</td>
<td>STAB SCBD+1</td>
</tr>
<tr>
<td>212</td>
<td>BF9F</td>
<td>BITA #$08</td>
</tr>
<tr>
<td>213</td>
<td>BF9F</td>
<td>BNE BAUDOK</td>
</tr>
<tr>
<td>214</td>
<td>BFA3 2609</td>
<td>* Change to div by 48 (5208 baud at 2 MHz)</td>
</tr>
<tr>
<td>215</td>
<td>BFA3</td>
<td>(equals: 8192 baud at 3.15 MHz)</td>
</tr>
<tr>
<td>216</td>
<td>BFA3</td>
<td>LDAB #$30</td>
</tr>
<tr>
<td>217</td>
<td>BFA6 C630</td>
<td>By default</td>
</tr>
<tr>
<td>218</td>
<td>BFA8</td>
<td>SLOBAUD EQU</td>
</tr>
<tr>
<td>219</td>
<td>BFA8</td>
<td>STAB SCBD+1</td>
</tr>
<tr>
<td>220</td>
<td>BFAA CC15AB</td>
<td>LDD #$DELAIS</td>
</tr>
<tr>
<td>221</td>
<td>BFAA</td>
<td>STD TOC1</td>
</tr>
<tr>
<td>222</td>
<td>BFAD D16</td>
<td>BAUDOK EQU</td>
</tr>
<tr>
<td>223</td>
<td>BFAD</td>
<td>LDY #$RAMSTR</td>
</tr>
<tr>
<td>224</td>
<td>BFBE 18CE0080</td>
<td>Point to start of RAM</td>
</tr>
<tr>
<td>225</td>
<td>BFBE</td>
<td>WAIT EQU</td>
</tr>
<tr>
<td>226</td>
<td>BFBE</td>
<td>LDX TOC1</td>
</tr>
<tr>
<td>227</td>
<td>BFBE</td>
<td>EQX *</td>
</tr>
<tr>
<td>228</td>
<td>BFBE</td>
<td>LOOP</td>
</tr>
<tr>
<td>229</td>
<td>BFBE</td>
<td>CLR</td>
</tr>
<tr>
<td>230</td>
<td>BFBE</td>
<td>DEX</td>
</tr>
<tr>
<td>231</td>
<td>BFBE</td>
<td>BNE</td>
</tr>
<tr>
<td>232</td>
<td>BFBE</td>
<td>BRA</td>
</tr>
</tbody>
</table>

524
NEWONE EQU *  
LDAA SCDRL Get received data
STAA S00,Y Store to next RAM location
STAA SCDRL Transmit it for handshake
INY Point to next RAM location
CPY #RAMEN+1 See if past end
BNE WAIT If not, get another

STAR EQU *  
JMP RAMSTR ** Exit to start of RAM **

* Block fill unused bytes with zero

BSZ $BFD1-*

*******************************************************************************
* Boot ROM revision level in ASCII
* (ORG $BFD1) "B"
*******************************************************************************
* Mask set I.D. ($0000 for EPROM parts)
* (ORG $BFD2) FDB $0000
*******************************************************************************
* 711K4 I.D. - can be used to determine MCU type
* (note: $4B = K in ASCII)
* (ORG $BFD4)
*******************************************************************************
* VECTORS - point to RAM for pseudo-vector JUMPs
*******************************************************************************
### Symbol Table:

<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Value</th>
<th>Def. #</th>
<th>Line Number</th>
<th>Cross Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUDOK</td>
<td>BFAF</td>
<td>*00231</td>
<td>00207</td>
<td>00222</td>
</tr>
<tr>
<td>BEGIN</td>
<td>BF5C</td>
<td>00175</td>
<td>00293</td>
<td></td>
</tr>
<tr>
<td>CONFIG</td>
<td>003F</td>
<td>*00047</td>
<td>00116</td>
<td></td>
</tr>
<tr>
<td>CONTINU</td>
<td>BF6A</td>
<td>*00186</td>
<td>00183</td>
<td></td>
</tr>
<tr>
<td>CYCLCOD</td>
<td>BE40</td>
<td>*00076</td>
<td>00185</td>
<td></td>
</tr>
<tr>
<td>DDRD</td>
<td>0009</td>
<td>*00032</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELAYF</td>
<td>0356</td>
<td>*00071</td>
<td>00192</td>
<td></td>
</tr>
<tr>
<td>DELAYS</td>
<td>15AB</td>
<td>*00070</td>
<td>00229</td>
<td></td>
</tr>
<tr>
<td>DONEIT</td>
<td>BF4F</td>
<td>*00163</td>
<td>00147</td>
<td></td>
</tr>
<tr>
<td>EEPMEND</td>
<td>OFFF</td>
<td>*00060</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEPMSTR</td>
<td>0D80</td>
<td>*00059</td>
<td>00203</td>
<td></td>
</tr>
<tr>
<td>ELAT</td>
<td>0020</td>
<td>*00042</td>
<td>00148</td>
<td>00151</td>
</tr>
<tr>
<td>EPGM</td>
<td>0001</td>
<td>*00043</td>
<td>00151</td>
<td></td>
</tr>
<tr>
<td>EPPMEND</td>
<td>7FFF</td>
<td>*00063</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPRMEND</td>
<td>2000</td>
<td>*00062</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPRMSTR</td>
<td>0002B</td>
<td>*00040</td>
<td>00149</td>
<td>00152 00162</td>
</tr>
<tr>
<td>INIT</td>
<td>BF12</td>
<td>*00114</td>
<td>00098</td>
<td>00137</td>
</tr>
<tr>
<td>NEWONE</td>
<td>BFBE</td>
<td>*00242</td>
<td>00237</td>
<td></td>
</tr>
<tr>
<td>NOTZERO</td>
<td>BFBE</td>
<td>*00204</td>
<td>00202</td>
<td></td>
</tr>
<tr>
<td>OCIF</td>
<td>00B0</td>
<td>*00038</td>
<td>00158</td>
<td>00161</td>
</tr>
<tr>
<td>PORTB</td>
<td>0004</td>
<td>*00029</td>
<td>00182</td>
<td></td>
</tr>
<tr>
<td>PORTD</td>
<td>0008</td>
<td>*00031</td>
<td>00196</td>
<td></td>
</tr>
<tr>
<td>PORTF</td>
<td>0005</td>
<td>*00030</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPROM</td>
<td>003B</td>
<td>*00045</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRGROUT</td>
<td>BF1D</td>
<td>*00136</td>
<td>00086</td>
<td></td>
</tr>
<tr>
<td>PROGDEL</td>
<td>1068</td>
<td>*00073</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM</td>
<td>BF00</td>
<td>*00086</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAMEND</td>
<td>037F</td>
<td>*00066</td>
<td>00176</td>
<td>00247</td>
</tr>
<tr>
<td>RAMSTR</td>
<td>0080</td>
<td>*00065</td>
<td>00232 00251</td>
<td></td>
</tr>
<tr>
<td>SCBD</td>
<td>0070</td>
<td>*00049</td>
<td>00189 00220</td>
<td>00228</td>
</tr>
<tr>
<td>SCCR1</td>
<td>0072</td>
<td>*00050</td>
<td>00191</td>
<td></td>
</tr>
<tr>
<td>SCCR2</td>
<td>0073</td>
<td>*00051</td>
<td>00195</td>
<td>00197</td>
</tr>
<tr>
<td>SCDRH</td>
<td>0076</td>
<td>*00054</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCDRL</td>
<td>0077</td>
<td>*00055</td>
<td>00103 00141</td>
<td>00145 00166 00200 00243 00245</td>
</tr>
<tr>
<td>SCCR1</td>
<td>0074</td>
<td>*00052</td>
<td>00102 00139</td>
<td>00144 00164 00199 00237</td>
</tr>
<tr>
<td>SCCR2</td>
<td>0075</td>
<td>*00053</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOBAUD</td>
<td>BFA8</td>
<td>*00227</td>
<td>00211 00216</td>
<td></td>
</tr>
<tr>
<td>STAR</td>
<td>BFCD</td>
<td>*00250</td>
<td>00240</td>
<td></td>
</tr>
<tr>
<td>TINT</td>
<td>00DE</td>
<td>*00034</td>
<td>00136</td>
<td></td>
</tr>
<tr>
<td>TEST1</td>
<td>003E</td>
<td>*00046</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFLGI</td>
<td>0023</td>
<td>*00036</td>
<td>00159 00161</td>
<td></td>
</tr>
<tr>
<td>TOC1</td>
<td>0016</td>
<td>*00035</td>
<td>00157 00199</td>
<td>00230 00235</td>
</tr>
<tr>
<td>UPLOAD</td>
<td>BF03</td>
<td>*00087</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULOOP</td>
<td>BF05</td>
<td>*00100</td>
<td>00105</td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td>BF03</td>
<td>*00234</td>
<td>00248</td>
<td></td>
</tr>
<tr>
<td>WAIT1</td>
<td>BF27</td>
<td>*00143</td>
<td>00168</td>
<td></td>
</tr>
<tr>
<td>WTLOOP</td>
<td>BF5</td>
<td>*00236</td>
<td>00239</td>
<td></td>
</tr>
</tbody>
</table>

Errors: None

Labels: 47

Last Program Address: $BBFF

Last Storage Address: $0000

Program Bytes: $0100 256

Storage Bytes: $0000 0
Use of Stack Simplifies M68HC11 Programming

By Gordon Doughman

INTRODUCTION

The architectural extensions of the M6800 incorporated into the M68HC11 allow easy manipulation of data residing on the stack of the microcontroller unit (MCU). The M68HC11 central processing unit (CPU) automatically uses the stack for two purposes. Each time the CPU executes a branch to subroutine (BSR) or jump to subroutine (JSR) instruction, it pushes a return address onto the stack. This procedure allows the CPU to resume execution with the instruction following the BSR or JSR when the program returns from the subroutine. Second, just before the MCU executes an interrupt service routine, the CPU saves its register contents on the stack, allowing the registers to be restored when the CPU executes a return from interrupt (RTI) instruction at the end of the interrupt service routine. Two additional uses of the M68HC11 stack discussed in this application note are the storage of local or temporary variable values and subroutine parameter passing.

Using the stack for local variables and parameter passing provides the assembly language programmer with the following benefits. First, since a routine allocates storage space for local variables and parameters upon entry and releases the storage upon exit, the same temporary memory space can be reused by program routines that run in succession. This reuse can result in a substantial savings in the total amount of RAM required by a program.

Second, allocating a new set of local variables and parameters when entering a routine makes it both reentrant and recursive. Routines that possess these two properties can make a programmer's job much easier when debugging a program in a real-time, interrupt-driven environment.

Third, placing local variables and parameters on the stack helps to promote modular programming. Because all temporary storage required by a routine is allocated and deallocated by the program module itself, it can be easily detached from the main program for reuse or replacement.

The final major benefit of using the stack for local variables and parameters becomes apparent during the debugging process. Because a routine's local variables and parameters exist only while it is executing, it is very unlikely that one routine will accidentally modify the local variables and parameters of another routine. Once the programmer has written and debugged a routine, time can be spent finding logical errors and/or problems associated with the interaction of the different routines in a program.

The goal of this application note is to help the assembly language programmer understand the following topics: 1) the basic operation of the M68HC11 stack, 2) the concept of local and global variables, 3) subroutine parameter passing, and 4) use of the M68HC11 instruction set to support local variables and parameter passing.

The source code for the examples and the macros described in this application note can be obtained by calling the Motorola FREEWARE Bulletin Board Service (BBS) at (512) 891-3733. The FREEWARE BBS operates 24 hours a day, 7 days a week, except for maintenance. The BBS's format is 300-2400 BAUD, 8 data bits, 1 stop bit, and no parity.
M68HC11 STACK OPERATION

The M68HC11 supports a stack through the use of the CPU stack pointer (SP) register. The SP is a 16-bit register that points to an area of RAM used for stack storage. Because the SP is 16 bits wide, the stack can be located anywhere in the M68HC11 64 K byte address space. The SP contents are undefined at power-up and are normally initialized in the first few instructions of a program. Each time a byte is pushed onto the stack, the SP is automatically decremented. Therefore, the initial value loaded into the SP is usually the address of the last RAM location in a system. Thus, as more information is pushed onto the stack, the stack area grows downward (the SP points to lower addresses) in the memory map. The SP always contains the address of the next available location on the stack.

As previously mentioned, the stack on the M68HC11 is used automatically by the CPU hardware during subroutine calls/returns and during the servicing of interrupts. When a subroutine is called by a JSR or BSR instruction, the address of the instruction following the JSR or BSR is automatically pushed onto the stack. Since the M68HC11 only has an 8-bit data bus, two separate push operations are performed by the CPU hardware. During the first push operation, the low-order eight bits (b7 – b0) of the return address are placed on the stack. The second push operation places the high-order eight bits (b15 – b8) of the return address on the stack at the next lower address in memory. Performing the operation in this order leaves the 16-bit return address on the stack in the order that all 16-bit numbers are stored in memory, with the high-order eight bits at the lower address. After a JSR or BSR instruction, the stack appears as shown in Figure 1.

![Figure 1. Stack Contents after Executing a JSR or BSR Instruction](image)

Whenever an unmasked interrupt occurs, the contents of all CPU registers (with the exception of the SP itself) are pushed onto the stack as shown in Figure 2. After the registers are stacked, CPU execution continues at an address specified by the vector for the pending interrupt source. Upon completion of the interrupt service routine, the execution of an RTI instruction restores the previously saved CPU registers by pulling them off the stack in the reverse order in which they were pushed onto the stack. Since the entire state of the CPU is restored, execution resumes as if the interrupt had not occurred.
The M68HC11 instruction set contains instructions that allow the individual CPU registers to be pushed onto and pulled off the stack. For example, if the value contained in one of the CPU registers needs to be saved before a particular subroutine call, a push instruction places the register value on the stack. When the subroutine returns, a pull instruction restores the contents of the CPU register. These instructions not only allow the stack to be used as temporary data storage but also allow the construction of recursive and reentrant subroutines. M68HC11 instructions that involve the direct manipulation of the SP are listed in Table 1.

Table 1. Instructions Involving Direct Manipulation of the SP

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSHA</td>
<td>Push Accumulator A onto the Stack.</td>
</tr>
<tr>
<td>PSHB</td>
<td>Push Accumulator B onto the Stack.</td>
</tr>
<tr>
<td>PULA</td>
<td>Pull Accumulator A off the Stack.</td>
</tr>
<tr>
<td>PULB</td>
<td>Pull Accumulator B off the Stack.</td>
</tr>
<tr>
<td>PSHX</td>
<td>Push Index Register X onto the Stack.</td>
</tr>
<tr>
<td>PSHY</td>
<td>Push Index Register Y onto the Stack.</td>
</tr>
<tr>
<td>PULX</td>
<td>Pull Index Register X off the Stack.</td>
</tr>
<tr>
<td>PULY</td>
<td>Pull Index Register Y off the Stack.</td>
</tr>
<tr>
<td>INS</td>
<td>Increment the Stack Pointer by 1.</td>
</tr>
<tr>
<td>DES</td>
<td>Decrement the Stack Pointer by 1.</td>
</tr>
<tr>
<td>TXS</td>
<td>Place the Contents of Index Register X - 1 in the Stack Pointer.</td>
</tr>
<tr>
<td>TYS</td>
<td>Place the Contents of Index Register Y - 1 in the Stack Pointer.</td>
</tr>
<tr>
<td>TXS</td>
<td>Place the Contents of the Stack Pointer +1 in Index Register X.</td>
</tr>
<tr>
<td>TSY</td>
<td>Place the Contents of the Stack Pointer +1 in Index Register Y.</td>
</tr>
</tbody>
</table>

STACK USAGE

Although most assembly language programmers use the M68HC11 stack for subroutine return addresses, register contents during interrupt processing, and temporary CPU register storage, more powerful programming techniques can make additional use of the stack.
Most high-level language compilers for modern, block-structured, high-level languages make use of the stack for two additional functions: passing parameters and local or temporary variable storage. By borrowing some of these techniques, programmers can write assembly language programs that are much more reliable, easier to maintain, and easier to debug.

**VARIABLES IN ASSEMBLY LANGUAGE**

Computer programs rarely operate on data directly; instead, the program refers to variables. A variable is a physical location in computer memory that can be used to hold different values while the program runs. Variables usually have an identifier or name associated with them. Using names to refer to data contained in memory is much easier than trying to remember a long string of binary or hexadecimal numbers.

Besides a name and an address, variables may have several other attributes. Depending on the programming language, variable declarations may assign attributes to the variables restricting both the scope and extent of the variable. The scope of a variable is the range of program text in which a particular variable is known and can be used. The extent of a variable is the time during which a computer associates physical storage with a variable name.

In assembly language, the scope of variables is usually global — i.e., variables may be referenced throughout the text of a program. Though some assemblers may provide mechanisms to restrict the scope of declared variables, many assembly language programmers do not use these features. A programmer using assembly language usually declares variables by employing an assembler directive as shown in Listing 1. This method assigns fixed storage locations to the variables. The extent of variables declared this way is for the entire program execution — i.e., the storage locations assigned to the variables at assembly time remain allocated during the entire time the program is executing.

```assembly
ORG $10

STATION PMB 1 STATION NUMBER REGISTER.
DATA LP PMB 1 DATA TABLE POINTER REGISTER.
STANSP PMB 1 STATION BIT MASK REGISTER.
FUNCTION PMB 1 FUNCTION NUMBER REGISTER FOR MODE SET.
XTEMP PMB 2 X-REG. TEMPORARY STORAGE.
XTMP1 PMB 2 X-REG1 TEMPORARY STORAGE.
XTMP1 PMB 1 A-REGISTER TEMPORARY STORAGE.
COUNT PMB 1 COUNT USED DURING STATION POLLING LOOP.
'NONE SELECTED' PMB 1 'NONE SELECTED' REGISTER USED BY SSCHK.

Listing 1. Declaring Global Variables in Assembly Language
```

Further examination of the variable declarations in Listing 1 shows that several variables are used for intermediate calculation results or for temporary CPU register storage. This example is typical of the way many assembly language programmers allocate temporary storage. Each time they write a routine requiring temporary variable storage, they allocate an additional set of global variables. The
The use of this technique can lead to the inefficient use of RAM if there are many routines within a program requiring temporary storage.

In an effort to make more efficient use of the limited amount of RAM on single-chip MCUs, some programmers use a technique known as “variable sharing.” Listing 2 shows a portion of a listing using this technique. In this program, more than one routine shares the use of a single temporary variable. To keep track of which routines use which variables, each line, in addition to the variable declaration, contains a list of the routines using that particular variable. In small programs, it may not be too difficult to manage temporary variables this way; however, in large programs having hundreds or thousands of routines using temporary variables, it becomes impossible to keep track of which routines use which temporary variables at any given time.

```
*** variables - used by: ***
PTR0  RMB 2  main,readbuff,incbuff,AS
PTR1  RMB 2  main,BR,DU,MO,AS,EX
PTR2  RMB 2  EX,DU,MO,AS
PTR3  RMB 2  EX,MO,MO,AS
PTR4  RMB 2  EX,AS
PTR5  RMB 2  EX,AS,BOOT
PTR6  RMB 2  EX,AS,BOOT
PTR7  RMB 2  EX,AS
PTR8  RMB 2  AS
TMP1  RMB 1  main,hexbin,buffarg,termarg
TMP2  RMB 1  GO,MO,AS,LOAD
TMP3  RMB 1  AS,LOAD
TMP4  RMB 1  TR,MO,ME,AS,LOAD
```

**Listing 2. Declaring Global Variables in Assembly Language**

The sharing of temporary variable storage shown in Listing 2 can produce debugging problems that are extremely hard to find. The chances of having one routine unintentionally modify the temporary storage of another can become quite high in large programs. In interrupt-driven, real-time systems, the sharing of temporary variables by various routines can become disastrous. Consider the situation illustrated in Figure 3. Subroutine A and subroutine B both share the temporary variable Temp1. Initially, there seems to be no problem since subroutine A and subroutine B do not call one another. Yet, consider what happens if an interrupt occurs during the execution of subroutine A. Because of the interrupt, subroutine B is called indirectly through subroutine C. The execution of subroutine B causes any value placed in Temp1 by subroutine A before the interrupt to be overwritten! Because interrupts usually occur asynchronously to main program execution, the program may appear to operate properly most of the time and crash randomly, depending on when an interrupt occurs. This type of apparently random program failure can be almost impossible to find.
Though this example may seem overly simplistic, a program that contains hundreds or thousands of routines makes it nearly impossible to keep track of which subroutines are using what variables at any specific time, particularly if the main program and interrupt service routines share subroutines. The solution to this type of problem may seem simple — do not allow any subroutines to share globally declared temporary variables. This solution is acceptable provided enough RAM is available for all required temporary variables. A better solution to this problem can be found by examining the way modern, block-structured, high-level languages use temporary variables.

VARIABLES IN BLOCK-STRUCTURED HIGH-LEVEL LANGUAGES

Most block-structured, high-level languages, notably C and Pascal, provide the ability to limit both the scope and the extent of variables as part of the language definition. In both C and Pascal, the scope of a variable is local to the block in which it is declared. The scope of variables declared outside of a block (function or procedure) is usually global. These global variables are similar to the ones declared in the assembly language shown in Listing 1. They can be accessed by all routines within a program, and they remain in existence throughout the entire time the program executes. Listing 3 shows an example of how global variables are declared in C and Pascal.
Listing 3. Declaring Global Variables in High-Level Languages

Variables declared within a function or procedure have their scope limited to that function or procedure. The extent of these variables is also limited. These variables, known as local or automatic variables, come into existence when the functions or procedures that contain them are called. When a function or procedure finishes execution, the local variables disappear, and the memory locations occupied by them can be used again. Listing 4 shows an example of how local variables are declared in C and Pascal. In both examples, the variables i and j are local to procedure/function A and do not exist outside them.

Listing 4. Declaring Local Variables in High-Level Languages

There are several benefits of using local variables. First, the restricted life of local variables can result in memory savings. Since storage for local variables is allocated upon entry to a routine and released upon exit from a routine, the same temporary memory space can be used by many different program routines. If two routines are run in succession, each can use the same storage locations.

Second, since a new set of local variables is allocated each time the procedure or function is entered, it makes the routine both reentrant and recursive. A reentrant routine is one that allocates a new set of local variables upon entry. When complex programs are run in a real-time, interrupt-driven environment, the interrupt handlers may call the routine that was interrupted. Making routines reentrant can greatly simplify a programmer’s job during the debugging process in a real-time environment. The same properties that make a routine reentrant also makes a routine recursive. A recursive routine is one that can call itself.
Third, the use of local variables helps to promote modular programming. A program module is a self-contained program element that can be easily detached from the main program either for reuse in another program or for replacement. Since any storage space for local variables is allocated and deallocated by the program module itself, the module code can easily be copied from a single place within one program and reused in another program.

A fourth benefit of using local variables is evidenced during the debugging process. In complex programs, there may be hundreds or thousands of routines that have to interact with each other. Since local variables help isolate any changes made within a routine, debugging becomes a much simpler process. Once routines are written and debugged, the programmer does not have to worry about one routine accidentally modifying the local variables of another. Instead, time can be spent finding any logical errors and/or problems associated with the interaction of routines in the program.

Even with all the benefits provided by the use of local variables, there are some costs associated with their use. On the M68HC11, programs using local variables tend to be slightly larger and slower than programs using only global variables because the addressing modes required to access the local variables can make the instruction somewhat longer and may cause longer execution time. Given the benefits of using local variables, a slightly larger and slower program is usually well worth the cost.

The reusable memory storage for local variables is usually taken from the same memory space used for the MCU's hardware stack. Placing local variables on the hardware stack leaves them intact even if the routine using them is interrupted. The specifics of allocating, deallocating, and accessing local variables residing on the M68HC11 stack is discussed in USING THE M68HC11 STACK.

PASSING PARAMETERS

To make routines more flexible and to vary their actions each time they are called, different information must be passed to the routines. Generally, most assembly language programmers use the CPU registers to pass information to a subroutine. Using this technique is acceptable as long as the amount of information to be passed to the subroutine fits within the available CPU registers.

When the amount of information to be passed to a routine exceeds the space available in the CPU registers, the information can be passed in a set of global variables. This technique may be acceptable for some situations, but it can also cause problems that make debugging difficult. One problem with passing parameters in this manner is that it makes a routine non-reentrant. Referring to Figure 4, assume that subroutine A's parameters are passed in a set of global variables. If subroutine A is called either by the main program or by subroutine C as a result of an interrupt, the program will work correctly. If an interrupt occurs during the execution of subroutine A, the original parameters passed by the main program will be overwritten when subroutine C calls subroutine A. When the processor returns from the interrupt and resumes execution of subroutine A, it will be using incorrect parameter data, and the results passed back to the main program will most likely be incorrect.
Because interrupts usually occur asynchronously to main program execution, the program may appear to operate properly most of the time and crash randomly. This type of problem can be extremely difficult to locate and can make debugging of real-time, interrupt-driven systems very difficult. Passing the parameters on the stack completely solves this problem. When subroutine C calls subroutine A as a result of the interrupt, a new set of parameters is placed on the stack while the original parameters remain undisturbed. Figure 5 shows the state of the stack after an interrupt.

Figure 5. Stack State as a Result of an Interrupt
In addition to where parameters are passed, there is also an issue of how parameters are passed. Subroutine parameters can be passed either by value or by reference. When a parameter is passed by value, the parameter acts as a local variable whose initial value is provided by the calling routine. Any modification of the supplied value has no effect on the original data that was passed to the subroutine. Thus, a subroutine can import values but not export values by means of value parameters.

Passing a parameter by reference is one method used to pass results back to a calling subroutine. These types of parameters are known as variable parameters. When using variable parameters, the address of the actual parameter is passed to the subroutine rather than a value. The passed address can be a local variable of the calling routine or even the address of a global variable. Whenever a subroutine has to effect a permanent change in the values passed to it, the parameters must be passed by reference rather than by value.

Consider the following example in both C and Pascal that exchanges the value of two integers:

### Pascal

**Call By Value**

```pascal
procedure SwapInt (x,y:integer);
var
  Temp:integer;
begin
  Temp:=x;
  x:=y
  y:=Temp
end;
```

**Call By Reference**

```pascal
procedure SwapInt (var x,y:integer);
var
  Temp:integer;
begin
  Temp:=x;
  x:=y
  y:=Temp
end;
```

**Call Of "SwapInt" Using Either Method**

```pascal
program(output);
var
  z,w:integer;
begin
  z:=2;
  w:=4;
  SwapInt (z,w);
end;
```

### C

**Call By Value**

```c
void SwapInt (int x,y)
{
  int Temp;
  Temp=x;
  x=y
  y=Temp
}
```

**Call By Reference**

```c
void SwapInt (int *x,*y)
{
  int Temp;
  Temp=*x;
  *x=*y
  *y=Temp
}
```

**Call Of "SwapInt" Using Call by Reference**

```c
main( )
{
  int w,z;
  z=2;
  w=4;
  SwapInt (&z,&w);
}
```

### Listing 5. Passing Parameters by Reference and by Value

If the call-by-value routine were to be used in this example, the routine would not work as the programmer might expect. It would exchange the local values of `x` and `y` within the `SwapInt` routine, but it would have no effect on the actual variables in the routine's call statement. For the `SwapInt` routine to work properly, the routine must be declared so that the parameters are passed by reference rather than by value. As mentioned previously, passing a parameter by reference passes the address of the actual parameter. In the example in Listing 5, using the call-by-reference routine, the addresses of the variables `z` and `w` are passed to the `SwapInt` routine when it is called from the main program. This procedure allows the `SwapInt` routine to exchange the actual values of the variables passed to the routine.
FUNCTION/SUBROUTINE RETURN VALUES

Most subroutines or functions, if they are to perform a useful action in a program, will return one or more values to the calling routine. Any value or status can be returned using one of the three methods previously described. When a subroutine only needs to return a single value, one of the CPU registers is commonly used to pass the value back to the calling routine. This simple, safe technique allows the routine to remain reentrant. This method is used most often by C compilers to return a value from a function.

Similar to the situation that exists when passing parameters in the CPU registers, there may be times when a routine must return more information than will fit in the CPU registers. The information can be returned in a set of global variables; however, as previously described, this method poses the same problems as passing parameters in this manner. Returning results in global variables makes the routine non-reentrant and can cause the same debugging problems previously described.

A better way to return large amounts of data from a subroutine is to allocate the required amount of space on the stack either just before or just after pushing a routine's parameters onto the stack. This method possesses the same benefits of passing parameters on the stack — it makes the routine completely reentrant and self-contained. Most Pascal compilers return function values in this manner.

USING THE M68HC11 STACK

This section specifically discusses how to allocate, deallocate, and access both local variables and parameters residing on the M68HC11 stack. The programmer's model of the M68HC11 is shown in Figure 6. The following paragraphs briefly describe the CPU registers and their usage.

```
7 ACCUMULATOR A 0 7 ACCUMULATOR B 0
15 DOUBLE ACCUMULATOR D 0
15 INDEX REGISTER X 0
15 INDEX REGISTER Y 0
15 STACK POINTER 0
15 PROGRAM COUNTER 0
7 CONDITION CODE REGISTER S X H I N Z V C
```

Figure 6. M68HC11 Programmer's Model

537
The A and B accumulators are used to hold operands and the results of arithmetic and logic operations. These two 8-bit registers can be concatenated to form a single 16-bit D accumulator to support the M68HC11 16-bit arithmetic instructions. The A and B accumulators can easily be used to push data onto or pull data off the stack.

The X and Y index registers are used in conjunction with the CPU indexed addressing mode. The indexed addressing mode uses the contents of the 16-bit index register in addition to a fixed 8-bit unsigned offset that is part of the instruction to form the effective address of the operand to be used by the instruction. The index registers play a very important role in accessing data residing on the stack.

The CPU SP is a 16-bit register that points to an area of RAM used for stack storage. The stack is used automatically during subroutine calls to save the address of the instruction that follows the call. When an interrupt occurs, the stack is used automatically by the CPU to save the entire CPU register contents on the stack (except for the SP itself). The SP always contains the address of the next available location on the stack.

The program counter (PC) is a 16-bit register used to hold the address of the next instruction to be executed.

The condition code register (CCR) contains five status indicators and two interrupt mask bits. The status bits reflect the results of arithmetic and other operations of the CPU as it performs instructions.

Before considering the specifics of parameter passing and the utilization of local variables that reside on the M68HC11 stack, the method used to access the information placed on the stack will be discussed. One M68HC11 index register and the CPU indexed addressing mode are used to access parameters or local variables residing on the stack. With respect to the indexed addressing mode, the contents of one of the 16-bit index registers plus a fixed unsigned offset is used in calculating the effective address of an instruction's operand. The unsigned offset, contained in a single byte following the instruction opcode, can only accommodate positive offsets in the range 0 – 255. Thus, the indexed addressing mode can only access information at addresses that are between 0 and 255 bytes greater than the base address contained in one of the index registers. Figure 7 illustrates how to calculate the effective address of an instruction using the indexed addressing mode.

![Figure 7. Effective Address Calculation for the Indexed Addressing Mode](image-url)
As information is pushed onto the M68HC11 stack, the SP is decremented, signifying that the information placed on the stack resides at addresses greater than the address contained in the SP. The use of indexed addressing is ideal for accessing information residing on the M68HC11 stack. The example shown in Figure 8 illustrates how information on the stack is manipulated.

![Figure 8. Stack Data Access Example](image)

As Figure 8 shows, the SP is pointing to the next available address, and the Y index register is pointing to the last data placed on the stack. The instruction `LDD 1, Y` will load the value of the local variable 'x' into the D accumulator. To access the parameter 'Num,' the instruction `LDD 7, Y` can be used. Any instructions that support the indexed addressing mode can be used to manipulate stack data.

**PASSING PARAMETERS**

Parameters are easily placed on the M68HC11 stack by CPU push instructions. Table 2 lists the push instructions available on the M68HC11. Note that there is not a single instruction for pushing the D accumulator onto the stack. A PSHD instruction can easily be simulated by executing the two instructions PSHB, PSHA. These two instructions must be executed in this order to keep the value pushed onto the stack consistent with the way 16-bit values are stored in memory — i.e., 16-bit values are placed in memory with the most significant eight bits at a lower address than the least significant eight bits. By following this convention, a 16-bit parameter pushed onto the stack in this manner is easily retrieved using one of the 16-bit load instructions.

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSHA</td>
<td>Push Accumulator A onto the Stack.</td>
</tr>
<tr>
<td>PSHB</td>
<td>Push Accumulator B onto the Stack.</td>
</tr>
<tr>
<td>PSHX</td>
<td>Push Index Register X onto the Stack.</td>
</tr>
<tr>
<td>PSHY</td>
<td>Push Index Register Y onto the Stack.</td>
</tr>
</tbody>
</table>

As previously mentioned, parameters can be passed either by value or by reference. Consider a function, Int2Asc, that converts a signed 16-bit integer to ASCII text and places the ASCII characters in a text buffer. The function requires two parameters: the number to be converted into ASCII text and...
a pointer to a buffer where the ASCII text is to be stored. The first parameter is passed to the subroutine by value because the actual number to be converted is passed to the function. The second parameter is passed by reference because a pointer to the buffer is passed to the routine and not the buffer itself. A function declaration written in C is shown in Listing 6.

```c
void Int2Asc(int Num; char *Buff)
{
    int Pwrlield 10 = 10000;
    char zs = 0;
    ...
    
}
```

Listing 6. Function Declaration of Int2Asc

Before calling an equivalent routine written in M68HC11 assembly language, the two parameters will be pushed onto the stack as shown in Listing 7.

```assembly
LDX ErrorNum
; Get the value of the current error.
PSHZ ; Place it on the stack.
LDX #OutBuff
; Get the address of the output buffer.
PSHZ ; Place it on the stack.
JSR Int2Asc
; Go convert the number.
```

Listing 7. Placing Parameters on the M68HC11 Stack

Using the immediate addressing mode with the second load index register X (LDX) instruction loads the address of OutBuff into the X index register rather than the 16-bit value contained in the memory locations OutBuff and OutBuff+1. After both parameters have been pushed onto the stack, the function is called with a JSR instruction. Upon entry to the subroutine Int2Asc, the parameters reside just above the return address as shown in Figure 9.

```
Figure 9. Location of Parameters Passed on the Stack
```

ALLOCATING LOCAL VARIABLES

Four basic techniques can be used to allocate local variables that reside on the stack. Choosing which one to use depends upon the total amount of storage required for the local variables and whether or
not the variables need to have an initial value assigned to them. Of course, a combination of all four techniques can be used.

One technique used to allocate space on the stack for local storage involves the use of the decrement stack pointer (DES) instruction. The DES instruction subtracts one from the value of the SP each time the instruction is executed, allocating one byte of local variable storage for each DES instruction. This technique is a simple and direct way of allocating local storage but becomes impractical when large amounts of local storage are required. For instance, if 100 bytes of local storage are required for a subroutine, 100 DES instructions are needed to allocate the required amount of storage. This required amount is clearly unacceptable since each DES instruction requires one byte of program memory. Even if a small program loop is set up to execute 100 DES instructions, the subroutine will suffer a severe execution speed penalty each time the routine is entered.

Using the previously described technique requires one byte of program storage for each byte of local storage that is allocated. Since allocating local storage simply involves decrementing the SP, the PSHX instruction can be used to allocate two bytes of local storage space for each executed PSHX instruction. The actual contents of the X index register are irrelevant because the only concern is decrementing the SP. The use of this technique can be confusing if not properly documented since it is not directly obvious what is being accomplished with five or six sequentially executed PSHX instructions.

Many times it is necessary to initialize local variables with a particular value before they are used. The same technique used to push parameters onto the stack before a subroutine call can also be used to allocate space for local variables and simultaneously assign initial values to them. This procedure is accomplished by loading one of the CPU registers with a variable's initial value and executing a PSH instruction. The program fragment in Listing 8 shows the use of this technique to allocate and initialize both an 8- and 16-bit local variable.

```
Int2Asc  equ  *
  .
  .
  idx   #10000   ; get the initial value of Pwr10.
  pshx   ; allocate and initialize it.
  clra   ; initial value of zs is zero.
  psha   ; allocate and initialize it.

Listing 8. Allocating and Initializing Local Variables
```

If more than 13 bytes of local storage are required by a subroutine, a fourth technique allocates storage more efficiently than using multiple DES or PSHX instructions. Since there are not any instructions that allow arithmetic to be performed directly on the SP, the fourth technique involves using several M68HC11 instructions. These instructions adjust the value of the SP downward in memory, allocating the required amount of local storage. Listing 9 shows the instruction sequence required to allocate an arbitrary number of bytes of local storage.

```
SinCos  equ  *
  .
  .
  tsx   ; SP+1 → X.
  xgd   ; exchange the contents of x and d.
  subd  #xxxx   ; subtract the required amt. of storage.
  xgd   ; place the result back into x.
  txx   ; X-1 → SP. Update the SP.

Listing 9. Allocation of More Than 13 Bytes for Local Storage
```
Since no single instruction allows the contents of the SP to be transferred to the D accumulator, the two-instruction sequence transfer from SP to index register X or Y; exchange double accumulator and index register X or Y (TSX; XGDX or TSY; XGDY) must be used. Placing the SP value in the D accumulator allows the use of the 16-bit subtract instruction to adjust the value of the SP. The subtract double accumulator (SUBD) instruction will subtract the 16-bit xxxx from the contents of the D accumulator. To place this new value in the SP, the two-instruction sequence XGDX; TXS or XGDY; TYS is used.

NOTE

Actually the TSX or TSY instruction causes the SP value plus one to be transferred to either the X or Y index register (SP + 1 → X or SP + 1 → Y). This transfer does not pose a problem because when the SP is updated with the TXS or TYS instruction, one is subtracted from the value of the index register (X − 1 → SP or Y − 1 → SP) before the SP is updated. Remember that since the SP points to the next available location on the stack, adding one to its value before the execution of the TSX or TSY instruction makes the X or Y index register point to the last data placed on the stack.

CREATING A COMPLETE STACK FRAME

In addition to providing storage space for local variables and parameters, a complete stack frame (sometimes called an activation record) must contain two additional pieces of information: a return address and a pointer to the base of the stack frame of any previous routines. The return address is placed on the stack automatically by the M68HC11 when it executes either a JSR or BSR instruction. As shown in Figure 9, the return address is placed on the stack just below a subroutine's parameters.

Before using either the X or Y index register to access a routine's parameters or local variables, the contents of the register must first be saved. The index register contents, known as the stack frame pointer, may contain the base address of a stack frame for a routine from which control was transferred. This pointer must be maintained so that when control is returned to the calling routine, the calling routine's environment can be restored to its previous state. Even if a routine has no local variables or parameters, the contents of the index register being used as the stack frame pointer must be saved before the register is used for any other purpose.

The best time to save the value of the previous stack frame pointer is immediately upon entry to a subroutine, which places the previous stack frame pointer immediately below the return address as shown in Figure 10.
After space for local variables has been allocated, the stack frame pointer for the new subroutine needs to be initialized. By transferring the contents of the SP to either the X or Y index register using the TSX or TSY instruction, a new stack frame is created.

In summary, creating a complete stack frame involves the following three steps after entering a subroutine:

1. Immediately upon entry to a subroutine, the contents of the index register being used as the stack frame pointer must be saved by using either the PSHX or PSHY instruction.
2. Storage space for the routine’s local variables should be allocated using one of the three methods described earlier.
3. The new stack frame pointer must be initialized using either the TSX or TSY instruction.

The last issue to discuss is which index register to use as the stack frame pointer. In terms of code size and speed, the X index register would be the most logical choice since all instructions involving the Y index register require one additional opcode byte and one additional clock cycle to execute. However, if a program is not making extensive use of the stack for local variables and parameters but is performing extensive array or table manipulations, the Y index register may be a better choice. No matter which index register is used as the stack frame pointer, it should be, if at all possible, dedicated to that use throughout a program. Program debugging is much easier if the contents of a single index register can always be expected to point to the current stack frame.

ACCESSING PARAMETERS AND LOCAL VARIABLES

As mentioned in Using the M68HC11 Stack, local variables and parameters are accessed by using instructions that support the indexed addressing mode. The following list identifies the load and store instructions as well as all arithmetic and logic instructions that support indexed addressing. Because most M68HC11 instructions support indexed addressing, it is just as code efficient to manipulate local variables that reside on the stack as it is to manipulate global variables using direct or extended addressing. Figure 11(a) illustrates a complete allocation frame as used by a subroutine.

Using the indexed addressing mode to access data contained in a stack frame places a restriction on the combined size of local variables and parameters. Since the indexed addressing mode functions by adding an unsigned 8-bit offset to the contents of the 16-bit index register, the indexed addressing mode can only access information at addresses that are between 0 and 255 bytes greater than the base address contained in one of the index registers. Consequently, the maximum size of a single stack frame is restricted to 256 bytes. If no parameters are passed to a routine on the stack, then the entire 256 bytes are available for local variables. However, when parameters are passed on the stack, not only is the space occupied by the parameters unavailable for use as local variables, but the subroutine return address and previous stack frame pointer reduce the amount of available space by an additional four bytes.
In most embedded control applications that use the M68HC11 in the single-chip mode, this limit on the combined size of parameters and local variables for a single stack frame is rarely a concern since the amount of on-chip RAM is limited. Several techniques can be used to work around the limit imposed by the indexed addressing mode; however, they are extremely wasteful in terms of code space and execution speed.

**NOTE**

In reality, the amount of memory available for local storage in a single stack frame is 257 bytes. Because the M68HC11 is capable of loading and storing 16 bits of data with a single instruction, it is possible to access one byte beyond the contents of the index register plus the fixed offset of 255 with the 16-bit load and store instructions.

**DEALLOCATING THE STACK FRAME**

When a subroutine has completed execution, the stack space allocated for the stack frame must be released so the memory can be reused by subsequent subroutine calls. The deallocation of the stack frame includes not only the removal of the space occupied by the local storage, but also the restoration of the previous stack frame pointer and the removal of space occupied by any parameters that were passed to the subroutine.

The process of freeing the memory occupied by the stack frame is simply a matter of adjusting the value of the SP upward in memory. The SP must be adjusted upward by the same amount that it was adjusted downward when the space for the stack frame was allocated. Either of the following methods can be used to perform this task.

The most obvious way to perform the deallocation is to reverse the process used to allocate the storage. Removing the stack frame in this manner involves three basic steps. First, the storage occupied by any local variables must be removed from the stack area by using the reverse of one of the techniques described in **ALLOCATING LOCAL VARIABLES**. Alternately, the technique shown in Listing 10 can be used. This technique involves adjusting the value of the SP upward in memory by the same amount it was adjusted downward when the space was allocated.

Second, the previous stack frame pointer must be restored. Because the previous stack frame pointer is now on the top of the stack, the use of a pull index register X or Y from the stack (PULX or PULY) instruction is all that is needed to perform this operation. At this point, the return address is on the top of the stack. Simply executing a return from subroutine (RTS) instruction returns program execution to the instruction following the subroutine call.

After returning to the calling routine, any parameters that were pushed onto the stack before the subroutine call must now be removed. This places the burden of removing subroutine parameters on the calling routine rather than on the called routine. This method of removing subroutine parameters is perfectly acceptable and is the one most often used by C language compilers.

Removing the parameters can be as simple as a one-instruction operation. If the X or Y index register contains the address of the current stack frame pointer, simply executing a TXS or TYS instruction places the SP just below the stack frame pointer. If the X or Y index register does not contain the
address of the current stack frame pointer, an alternate method must be used to remove the parameters. Figure 11 illustrates the state of the stack at each stage of the deallocation process.

An alternate method requires the called routine to remove the entire stack frame, including any parameters passed to it. This method may not be as code efficient as the first method since it requires a fixed number of instructions to release the storage space occupied by the entire stack frame. Listing 11 shows the instruction sequence necessary to deallocate the stack frame when the X index register is being used as the stack frame pointer. This four-instruction sequence requires nine bytes of program storage space and 18 cycles to execute but removes the entire stack frame, regardless of the size. This method of stack frame deallocation has one drawback — the X or Y index register must always contain a valid stack frame pointer. Thus, all subroutines, even if they do not require parameters or local variables, must “mark” the current state of the stack upon entry by executing a PSHX; TSX or PSHY; TSY instruction sequence.
Figure 11. Deallocation of the Stack Frame

NOTE

In Listing 11, RA is the offset value to the <Return Address> and PSFP is the offset value to the <Previous Stack Frame Pointer>.

LDY RA,X  Load the return address into the Y register.
LDX PSFP,X  Restore the previous stack frame pointer.
TXS  Remove the entire stack frame.
JMP 0,Y  Return to the calling routine.

Listing 11. Alternate Method for Dealocating the Entire Stack Frame
In summary, choosing a method to deallocate the stack frame involves a tradeoff between code size and execution speed. Using the first method results in the smallest amount of code being generated but may take longer to execute than the method shown in Listing 11.

**SUPPORT MACROS**

The following macros may be used to help in managing stack frames in M68HC11 programs. Using these macros may not provide the smallest or fastest code in all situations but should make the program easier to write and debug. Although the macros were written for the Micro Dialects μASM-HC11™ assembler that runs on the Macintosh™, they can be used with other assemblers with some modification. The following paragraph explains the way parameters are passed and referenced in the Micro Dialects assembler and should help in the conversion process.

When a macro is defined, parameters are not declared. When a macro is invoked, the parameters appear in the operand field following the macro name. Within a macro definition, parameters are referenced by using a colon (:) followed by a single decimal digit (0–9). Therefore, within the body of the macro, the first parameter is referenced by using ':0', the second parameter is referenced by using ':1', and so forth. Parameter substitution is performed strictly on a textual substitution basis.

The link macro shown in Listing 12 can be used to allocate a complete stack frame after entry into a subroutine. The link macro performs the following functions: 1) saves the previous stack frame pointer, 2) allocates the required number of bytes of local storage, and 3) initializes a new stack frame pointer. The calling convention for the link macro is as follows:

```
link <s.f. reg>,<storage bytes>
```

The first parameter passed to the macro is the name of the index register being used as the stack frame pointer (either X or Y). Although no check is made to ensure that a legal index register name is passed to the macro, the assembler will produce an "Unrecognized Mnemonic" error message when the macro is expanded. The second parameter is the number of bytes of local storage required by the subroutine.

```
link .macro
  ps:0 ; Save the previous stack frame pointer.
  ts:0 ; Transfer the stack pointer into :0.
  xp:0 ; Transfer :0 into D.
  subd *=1 ; subtract the required amount of local storage.
  xpd:0 ; Initialize the new stack frame pointer
  t:0s ; Update the stack pointer with new value.
.endm
```

**Listing 12. The Link Macro**

The return and deallocate (rtd) macro shown in Listing 13 can be used to partially deallocate a subroutine stack frame. The rtd macro performs the following functions: 1) deallocates local storage, 2) restores the previous stack frame pointer, and 3) returns to the calling routine. The rtd macro DOES NOT remove any parameters from the stack that may have been passed to the subroutine. Removal of any parameters must be performed by the calling routine. This macro is useful when no parameters are passed to a subroutine or when parameters are passed in registers. The calling convention for the rtd macro is as follows:

```
rtd <s.f. reg>,<storage bytes>
```

μASM-HC11 is a trademark of Micro Dialects.
Macintosh is a trademark of Apple Computer, Inc.
Like the link macro, the first parameter passed to the rtd macro is the name of the index register being used as the stack frame pointer (either X or Y). Again, although no check is made to ensure that a legal index register name is passed to the macro, the assembler will produce an “Unrecognized Mnemonic” error message when the macro is expanded. The second parameter is the number of bytes of local storage allocated when the subroutine was entered.

```
rtd macro
  ldab #1 ; number of bytes to deallocate.
  ab:0 ; add it to the current stack frame pointer.
  t:0s ; deallocate storage by updating the stack pointer.
  pul:0 ; restore the previous stack frame pointer.
  rts ; return to the calling routine.
endm
```

**Listing 13. The Return and Deallocate Macro**

The only drawback in using this macro is that it uses the B accumulator when deallocating a subroutine’s local storage, preventing a subroutine from returning a 16-bit result in the D accumulator. A simple solution to the problem is to surround the load accumulator B (LDAB) and add accumulator B to index register X or Y (ABX/ABY) instructions with the PSHB/PULB instruction pair as shown in Listing 14. This macro, renamed frtd for function return and deallocate, allows the D accumulator to be loaded with a return value immediately before the macro is called. A second solution to this problem is to place all return values on the stack as described in FUNCTION/SUBROUTINE RETURN VALUES, allowing the calling routine to retrieve the returned value and then remove it along with the parameters.

```
frtd macro
  pshb ; save the lower byte of the return value.
  ldab #1 ; number of bytes to deallocate.
  ab:0 ; add it to the current stack frame pointer.
  pulb ; restore the lower byte of the return value.
  t:0s ; deallocate storage by updating the stack pointer.
  pul:0 ; restore the previous stack frame pointer.
  rts ; return to the calling routine.
endm
```

**Listing 14. The Function Return and Deallocate Macro**

The return and deallocate using x (rtdx) and return and deallocate using y (rtdy) macros shown in Listing 15 can be used to completely deallocate a subroutine stack frame, including any parameters that were passed on the stack. The rtdx and rtdy macros perform the following functions: 1) deallocates the entire stack frame, including local storage and passed parameters, 2) restores the previous stack frame pointer, and 3) returns to the calling routine. The calling convention for the rtdx and rtdy macros is as follows:

```
rtdx <storage bytes> or rtdy <storage bytes>
```

The only parameter passed to the macros is the number of local storage bytes allocated upon entry to the subroutine. These macros have an advantage over the rtd macro in that the A and B accumulators are not used during deallocation, which allows a return value to be loaded into the A, B, or D registers before execution of the rtdx or rtdy macro.
Listing 15. The rtdx and rtdy Macro

The only restriction to using the rtdx and rtdy macros is that a valid stack frame pointer for the previous subroutine must be present in either the X or Y index register when the register is pushed onto the stack at the beginning of the subroutine. Even if a subroutine has no local variables in it or no parameters passed to it, a PSHX and TSX instruction must be executed immediately upon entry to a subroutine to save the previous stack frame pointer and “mark” the current state of the stack. Before returning, a PULX instruction must be executed to restore the previous stack frame pointer.

This restriction implies that, somewhere in the program, the index register to be used as the stack frame pointer must be initialized with a valid value. If either the X or Y index register is to be dedicated for use as a stack frame pointer, the index register must be initialized at the beginning of the program. The initial value loaded into the index register should be one more than the value loaded into the stack pointer, which is easily accomplished by executing the TSX instruction immediately after initializing the stack pointer.

In summary, the use of the rtdx and rtdy macros are convenient in that they remove both parameters and local variables passed to subroutines. However, their use will cost three extra instructions in subroutines that do not have local variables or parameters but call subroutines that use local variables or have parameters passed to them.

EXAMPLES

Appendix A contains several examples that use the techniques described to manage local storage, parameter passing, and allocation/deallocation of stack frames.
APPENDIX A
EXAMPLE LISTINGS

Include "Stack Macros"

Written By
Gordon Doughman
For
Motorola Semiconductor

The author reserves the right to make changes to this file. Although this software has been carefully reviewed and is believed to be reliable, neither Motorola nor the author assumes any liability arising from its use. This software may be freely used and/or modified at no cost or obligation to the user.

The following macros may be used to help in managing stack frames in M68HC11 programs. The macros were written for Micro Dialects μASM-HC11 assembler that runs on the Macintosh but may be used with other assemblers with some modification. The following discussion of the way parameters are passed and referenced should help in the conversion process.

Within a macro, parameters are referenced by using a colon (:) followed by a single decimal digit (0-9). Therefore, within the body of the macro the first parameter is referenced by using ':0', the second parameter is referenced by using ':1', and so forth. Parameter substitution is performed strictly on a textual substitution basis.

The link macro may be used to allocate a complete stack frame after entry into a subroutine. The link macro performs the following functions:
1) Saves the previous stack frame pointer;
2) Allocates the requested number of bytes of local storage;
3) Initializes a new stack frame pointer.

Usage:
link <s.f. reg>,<storage bytes>

The first parameter passed to link is the index register that is being used as the stack frame pointer (either x or y). Although no check is made to ensure that a legal index register name is passed to the macro, the assembler will produce an "Unrecognized Mnemonic" error message when the macro is expanded. The second parameter is the number of bytes of local storage required by the subroutine.

link macro
psh:] ; Save the previous stack frame pointer.
lviu
M
xgd:0 ; Transfer :0 into D.
M
subd *:1 ; subtract the required amount of local storage.
xgd:0 ; Initialize the new stack frame pointer
M
t:0s ; Update the stack pointer with new value.
M
endm

The rtd (Return and Dealloace) macro may be used to partially deallocate a subroutine stack frame that includes parameters passed on the stack. The rtd macro performs the following functions: 1) Deallocates local storage;
2) Restores the previous stack frame pointer; 3) Returns to the calling routine. rtd DOES NOT remove any parameters from the stack. This function must be performed by the calling routine. This macro is useful when parameters are passed in registers rather than on the stack.

Usage:
rtd <s.f. reg>,<storage bytes>

The first parameter passed to link is the index register that is being used as the stack frame pointer (either x or y). Although no check is made to ensure that a legal index register name is passed to the macro, the assembler will produce an "Unrecognized Mnemonic" error message when the macro is expanded. The second parameter is the number of bytes of local storage
The ftd (Function Return and Deallocate) macro may be used to partially deallocate a subroutine stack frame that includes parameters passed on the stack. The ftd macro performs the following functions: 1) Deallocation of local storage; 2) Restores the previous stack frame pointer; 3) Returns to the calling routine. The ftd macro DOES NOT remove any parameters from the stack. This function must be performed by the calling routine. This macro is useful when parameters are passed in registers rather than on the stack and a value is being returned in the D-accumulator.

Usage:

frtd <s.f. reg>,<storage bytes>

The first parameter passed to ftd is the index register that is being used as the stack frame pointer (either x or y). Although no check is made to ensure that a legal index register name is passed to the macro, the assembler will produce an "Unrecognized Mnemonic" error message when the macro is expanded. The second parameter is the number of bytes of local storage used by the subroutine.

The rtdx and rtdy (Return and Deallocate using x or y) macros may be used to completely deallocate a subroutine stack frame including parameters that were passed on the stack. The rtdx macro performs the following functions: 1) Deallocates the entire stack frame including local storage and passed parameters; 2) Restores the previous stack frame pointer; and 3) Returns to the calling routine.

Usage:

rtdx <storage bytes>
rtdy <storage bytes>

The only parameter passed to the routines is the number of bytes of local storage that were originally allocated upon entry to the subroutine. These macros have the advantage that the a and b accumulators are not used during the deallocation process. This allows a value to be loaded into the a, b, or d registers before the execution of the rtdx or rtdy macro and returned to the calling routine.
pshd macro
  ldx  0,x
  ldy  0,y
  tyl
  jmp  0,x
: load the return address into the x-index register.
: restore the previous stack frame pointer.
: Update the stack pointer, removing the storage space.
: Return to the calling routine.

******************************************************************************

* The pshd macro pushes the 16-bit d-accumulator onto the stack. The
* b-accumulator is pushed first so that the least significant 8-bits of
* the 16-bit number appear on the stack at the higher address. This is
* consistent with the way all 16-bit numbers are stored in memory.
* Usage:    pshd
* No parameters are required by the macro.
******************************************************************************

pshd   : macro
pushb
psha
: pushh
: endm

******************************************************************************

* The puld macro pulls the top two bytes from the stack and places them in
* the 16-bit d-accumulator. The first byte pulled from the stack is placed
* in the a-accumulator; the second byte pulled from the stack is placed in
* the b-accumulator. The pull order is consistent with the way all 16-bit
* numbers are stored in memory.
* Usage:    puld
* No parameters are required by the macro.
******************************************************************************

puld   : macro
pula
pulb
: pulb
: endm

******************************************************************************

* The clrd macro uses the clra and clrb instructions to clear the 16-bit
* d-accumulator.
* Usage:    clrd
* No parameters are required by the macro.
******************************************************************************

clrd   : macro
cira
clrb
: clrb
: endm
The author reserves the right to make changes to this file. Although this software has been carefully reviewed and is believed to be reliable, neither Motorola nor the author assumes any liability for its use. This software may be freely used and/or modified at no cost or obligation to the user.

This subroutine converts a 16-bit binary integer to a null terminated ASCII string. Three parameters are passed to the subroutine on the stack. The first parameter is the 16-bit binary number to be converted.

The second parameter is the address of a buffer where the null terminated ASCII string will be placed. The buffer should be at least 7 bytes long.

The third parameter is a boolean flag indicating whether the number passed in the first parameter is a signed or unsigned 16-bit number. If the byte flag is zero, the number is converted as an unsigned number. If the byte flag is non-zero, the number will be converted as a 16-bit signed number.

Parameters are pushed onto the stack in the following order: 1) Signed Flag; 2) Pointer to ASCII buffer; 3) Number to be converted. A typical calling sequence would be:

```
clra          ; Do the conversion as an unsigned number.
psha         ; put the flag on the stack.
ldd #Buffer   ; get the address of the ascii buffer.
pshd         ; put the address on the stack.
ldd Num       ; Get the number to convert.
pshd         ; Put it on the stack
jse Int2Asc   ; Go convert the number.
```

This subroutine has two local variables. The first, zs, is a boolean variable used to suppress leading zeros when doing a conversion. It is located at an offset of 0 from the stack frame pointer. The second local, Divisor, is a 16-bit variable. It is used to divide the number being converted by successively lower powers of 10. Divisor is located at an offset of 1 from the local stack frame pointer.

NOTE: This routine was written assuming that the previous stack frame pointer is the x-index register. HOWEVER, because the x-index register is required by the integer divide instruction, the y-index register is used as the stack frame pointer WITHIN the Int2Asc subroutine.

```
Declare locals

PCSave set * ; save the current PC value
org 0 ; set PC to 0 for offsets to locals
zs rmb 1 ; declare zs variable.
Divisor rmb 2 ; declare Divisor variable.
LocSize set * ; number of bytes of local storage.
org PCSave

Offsets to parameters

Num equ LocSize+4 ; offset to Num parameter.
BuffP equ LocSize+6 ; offset to BuffP parameter.
Signed equ LocSize+8 ; offset to Signed parameter.
Int2Asc equ * ; save the previous stack frame pointer.
pshx
psb
psa
clra
```

553
This subroutine performs a 16 x 16 bit unsigned multiply and produces a 32-bit result. Two 16-bit numbers are passed to the subroutine on the stack.

The 32-bit result is returned on the stack in place of the two 16-bit parameters. This allows the calling routine to easily pull the product from the stack and store the result. Because multiplication is a commutative operation, the order in which the parameters are pushed onto the stack is unimportant. A typical calling sequence would be:

```
* 1dd Num1
* pshd
* 1dd Num2
* pshd
* jbr Mul16x16
* puld
* std Result32
* puld
* std Result32+2
```
This subroutine has four local variables. Each variable occupies 1 byte on the stack. These four bytes are used to hold the partial product as the final answer is being computed. These four byte variables are treated as 16-bit variables during the calculation.

NOTE: This routine was written assuming that the stack frame pointer is the x-index register.

Declare locals

0073 PCSave set * ; save the current PC value
0000 org 0 ; set PC to 0 for offsets to locals
0000 Prd0 rmb 1 ; declare ms byte of partial product variable.
0001 Prd1 rmb 1 ; declare next ms byte of partial product variable.
0002 Prd2 rmb 1 ; declare next ls byte of partial product variable.
0003 Prd3 rmb 1 ; declare ls byte of partial product variable.
0004 LocSize set * ; number of bytes of local storage.
0073 PCSave org PCSave

Offsets to parameters

0008 Fact1 equ LocSize+4 ; offset to factor 1 parameter.
000A Fact2 equ LocSize+4 ; offset to factor 2 parameter.

cycles clear

Mul6x16 equ * ; save the previous stack frame pointer.

pshx ; clear the d-accumulator.

clrd

clrb

pshd ; allocate & initialize the locals prd0 - prd3

psb

psa

psb

psa

psb

psa

psb

tsx ; initialize the new stack frame pointer.

ldaa Fact1+1,x ; get the ls byte of factor 1.

ldab Fact2+1,x ; get the ls byte of factor 2.

mul ; multiply them.

std Prd2,x ; save the first term of the partial product.

ldaa Fact1,x ; get the ms byte of factor 1.

ldab Fact2+1,x ; get the ls byte of factor 2.

mul ; multiply them.

std Prd1,x ; save the result.

add Prd1,x ; add the result into the partial product.

std Prd1,x ; save the result.

std Prd2,x ; save the result.

mul ; multiply them.

std Prd2,x ; save the result.

add Prd0,x ; add it to the partial product.

std Fact1,x ; overwrite the two parameters with the result.

ldd Prd2,x

std Fact2,x

rtd x,LocSize ; return and deallocate the locals.

ldab #LocSize ; number of bytes to deallocate.

abx ; add it to the current stack frame pointer.

txs ; deallocate storage by updating the stack pointer.

pulx ; restore the previous stack frame pointer.

rts ; return to the calling routine.

cycles total=170 ; Total number of E cycles for a 16 x 16 multiply.

555
This subroutine performs a 32 by 16 bit unsigned divide and produces a 32-bit quotient and a 16-bit remainder. Both the divisor and dividend are passed to the subroutine on the stack. The 32-bit quotient and 16-bit remainder are returned on the stack in place of the divisor and dividend. This allows the calling routine to easily pull the answer from the stack and store the result. The divisor is pushed onto the stack first, followed by the lower 16-bits of the dividend and finally the upper 16-bits of the dividend. A typical calling sequence would be:

```
ldd Divisor
pshd
ldd Dividend+2
pshd
ldd Dividend
pshd
jsr Div32x16
puld std Quotient
puld std Quotient+2
puld std Remainder

```

This subroutine has two local variables. A 32-bit variable for partial quotient results that is treated as two 16-bit variables and a 16-bit variable for intermediate remainder results.

NOTE: This routine was written assuming that the previous stack frame pointer is the x-index register. HOWEVER, because the x-index register is required by the integer and fractional divide instructions, the y-index register is used as the stack frame pointer WITHIN the Div32x16 subroutine.

Declare locals

```
PCSave set * ; save the current PC value.
org 0 ; set PC to 0 for offsets to locals.
Quo0 rmb 2 ; declare upper 16-bits of quotient.
Quo2 rmb 2 ; declare lower 16-bits of quotient.
LocSize set * ; number of bytes of local storage.
org PCSave

```

Offsets to parameters

```
Num0 equ LocSize+4 ; upper 16-bits of Dividend.
Num2 equ LocSize+6 ; lower 16-bits of Dividend.
Denm equ LocSize+8 ; 16-bit divisor.

```

cycles clear

```
Div32x16 equ * ; save the previous stack frame pointer.

```

```
00AC clrd ; clear the d-accumulator.
00AF pshd ; allocate & initialize the locals.
00AE 37 pshb
00AF 36 psha
00BD pshd
00BE 37 pshb
00BF 36 psha
00B2 pshd
00B3 36 psha
00B4 1830 tsy ; initialize y as the new stack frame pointer.
00B6 18BC0A [4] ldd Num0,y
00B9 CDA30E [7] cpd Denm,y ; is the divisor=the upper 16-bits of the dividend?
00BC 2507 [3] blo Div32x16a ; yes. use a fractional divide on the initial value.

556
506 00BE CDEEE0 [ 6] ldx Denm,y 
507 00C1 02 [41] ldiv 
508 00C2 CDEFF0 [ 6] stx Quo0,y 
509 00C5 CDEEE0 [ 6] Div32x16a ldx Denm,y 
510 00C8 03 [41] fdiv 

511 00C9 CDEFF2 [ 6] stx Quo2,y 
512 00CC :8ED04 [ 6] std Rem,y 

513 00CF 18EC0C [ 6] ldd Num2,y 
514 00D2 CDEEE0 [ 6] ldx Denm,y 
515 00D5 02 [41] idiv 
516 00D6 18E304 [ 7] add Rem,y 
517 00D9 18ED04 [ 6] std Rem,y 
518 00DC 8F [ 3] xgdx 

519 
520 00DD 18E302 [ 7] addd Quo2,y 

521 00E0 18ED0C [ 6] std Num2,y 
522 00E3 18EC00 [ 6] ldd Quo0,y 
523 00E6 C900 [ 2] adcb #0 
524 00E8 8900 [ 2] adca #0 
525 00EA 18E00A [ 6] std Num0,y 
526 00ED 8F [ 3] xgdx 
527 00EE CDA30E [ 7] cmpd Denm,y 
528 00F1 2519 [ 3] blo Div32x16b 
529 00F3 18A30E [ 7] subd Denm,y 
530 00F6 18ED04 [ 6] std Rem,y 
531 00F9 18EC0C [ 6] ldd Num2,y 
532 00FC C30001 [ 4] addd #1 
533 00FF 18ED0C [ 6] std Num2,y 
534 0102 18EC0A [ 6] ldd Num0,y 
535 0105 C900 [ 2] adcb #0 
536 0107 8900 [ 2] adca #0 
537 0109 18ED0A [ 6] std Num0,y 
538 010C 18EC04 [ 6] Div32x16b ldd Rem,y 
539 010F 18ED0E [ 6] std Denm,y 
540 0112 30 [ 3] txs 

541 0113 rtd x,LocSize 
544 0116 35 [ 3] txs 
545 0117 38 [ 5] pux 
546 0118 39 [ 5] rts 

547 * 
548 cycles total=347 
549 * 

Errors: None 
Labels: 30 
Last Program Address: S0118 
Last Storage Address: SFFFF 
Program Bytes: S0119 281 
Storage Bytes: S000D 13 

Total number of E cycles for a 32 x 16 divide.
Use of the MC68HC68T1 Real-Time Clock with Multiple Time Bases

Prepared by: Jim Carlson
Field Applications Engineering

INTRODUCTION

The MC68HC68T1 is an HCMOS peripheral chip containing a real-time clock/calendar and a 32 byte static RAM array. A synchronous, serial, three-wire interface (synchronous peripheral interface) is provided for communication with a microcomputer. The MC68HC68T1 can use a crystal or the 50/60 hertz line frequency as its timebase.

Often applications are line powered during normal operation but must continue to maintain the correct time-of-day when powered down. During normal operation the timebase is provided by the power mains (50 or 60 hertz). When the main power is lost, the 50/60 hertz timebase is lost. At this point, the clock must switch to crystal operation to provide correct time keeping. The MC68HC68T1 is not directly capable of this; additional support must be provided by the host microcomputer. This application note makes use of the Motorola MC68HC11ABP1 as the host MCU, as it provides the necessary Serial Peripheral Interface. MCUs that do not provide this functionality may emulate it in software by manipulating parallel I/O.

The power line is chosen as the preferred timebase because of the accuracy provided by the electric utility companies. A 32.768 kHz crystal, on the other hand, is quite sensitive to temperature variations, and to the voltage on which the oscillator runs. Temperature sensitivity often exceeds 0.035 ppm/°C and aging can exceed 5 ppm/year. This may cause errors exceeding twenty seconds per month, if the crystal is subjected to wide temperature variations, such as encountered in remote, outdoor sites.

Contrast this error with that achieved with the power line as a timebase, which may be under one second per month.

This application note discusses the transition from line to crystal timebase operation during line power failure, and back to line power timebase when line power is reestablished.

CIRCUIT OPERATION

The purpose of this circuit is to demonstrate the use of the MC68HC68T1 in a application using both the power line and a crystal as the timebase, depending on the availability of line power. As such, it is just a skeleton of an actual system. No method to set the clock is provided; just a subroutine called "SET_CLOCK" which always sets the clock to 00:00:00, Friday, January 1, 1990 (24 hour time). An actual application would, of course, prompt the operator for the correct time and date.

The demonstration software puts the MCU into a continuous loop, where the time (hours, minutes, and seconds), maintained in the MC68HC68T1 is read, converted into an ASCII representa-
outputs of interest here are the Power Warning and the Reset outputs. The Reset output generates a reset to the MCU when power is applied to the system. The Power Warning output generates an interrupt to the MCU when the unregulated voltage drops below a value considered to be dangerously low, but not yet below the minimum operating voltage of the regulator. Please refer to Figure 2.

An uncommitted voltage comparator is also provided by the MC34160, and is used to convert the 60 Hz sine wave from the power transformer into a logic signal to drive LINE on the MC68HC68T1. Since this comparator provides hysteresis, noise immunity is improved.

When the Power Warning output generates an interrupt to the MCU, the MCU completes the current instruction, or series of instructions in the critical code section, and then enters the interrupt handler code. This code simply reconfigures the MC68HC68T1 to crystal timebase operation by writing the value %111110100 to the Clock Control Register. Please note that only bit D6 was changed, as compared to the value previously written. The interrupt handler routine would then save whatever other information was required by the application into the MC68HC68T1's internal RAM. In this example, however, nothing needs to be saved, so the MCU just enters the STOP state.

The code section that reads the time information from the MC68HC68T1 is considered a critical code section, because it accesses MCU resources which are also accessed from within the interrupt handler. If the interrupt were to be accepted while serial transmissions were in progress, errors would occur, because the interrupt handler would not allow the completion of the transmission, but would simply start its own transmission. This could cause a "Write Collision" in the S.P.I., or erroneous data to be sent to the MC68HC68T1. This is prevented by simply bracketing this critical code with instructions that disable interrupts at the beginning, and enable interrupts at the end.

The value of the power supply filter capacitor (Cfilter in Figure 1) must be chosen so that Tsave (Figure 2) is long enough to allow the MCU to completely execute the interrupt handler code before the unregulated voltage drops to a point below which the regulator will not operate. The value of Cfilter (Figure 1) provides approximately 150 ms of operation after the interrupt is generated.

The MC34164P Low Voltage Detect Circuit guarantees that the MCU is held in the reset state when the supply voltage drops below the level that allows the MC34160 to function correctly. This prevents possible program runaway which may modify registers in the MC68HC68T1.

The program source code listing is included. This software was written for Motorola's MC68HC11 cross macro assembler.

REFERENCES

"CX–IV Crystal Data Sheet," Statek Corporation, 12/87.
This software demonstrates the use of the MC68HC68T1 Real Time Clock in a system that normally derives its timebase from the ac line, but during a power failure, switches to a 32.768 kHz crystal.

This software runs on an MC68HC11A8P1 microcomputer. This MCU contains a debug monitor called "BUFFALO". This software is stored in the MCU's on-chip EEPROM.

The purpose of this program is to output the time-of-day to a CRT terminal connected to the MCU's SCI (UART) port. When a power failure occurs, the MCU is interrupted with an IRQ, then it switches the real time clock to crystal control. When power is reapplied, the MCU switches the real time clock back to line power timebase.

**MC68HC68T1 Register Addresses:**

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seconds</td>
<td>$20</td>
</tr>
<tr>
<td>Minutes</td>
<td>$21</td>
</tr>
<tr>
<td>Hours</td>
<td>$22</td>
</tr>
<tr>
<td>Day</td>
<td>$23</td>
</tr>
<tr>
<td>Date</td>
<td>$24</td>
</tr>
<tr>
<td>Month</td>
<td>$25</td>
</tr>
<tr>
<td>Year</td>
<td>$26</td>
</tr>
<tr>
<td>Sec_Alarm</td>
<td>$27</td>
</tr>
<tr>
<td>Min_Alarm</td>
<td>$28</td>
</tr>
<tr>
<td>Status</td>
<td>$29</td>
</tr>
<tr>
<td>First_Up</td>
<td>$30</td>
</tr>
<tr>
<td>Control</td>
<td>$31</td>
</tr>
<tr>
<td>Start</td>
<td>$32</td>
</tr>
<tr>
<td>Line</td>
<td>$33</td>
</tr>
<tr>
<td>XTL_SEL1</td>
<td>$34</td>
</tr>
<tr>
<td>XTL_SEL0</td>
<td>$35</td>
</tr>
<tr>
<td>Line_50</td>
<td>$36</td>
</tr>
<tr>
<td>Interrupt</td>
<td>$37</td>
</tr>
<tr>
<td>Write</td>
<td>$38</td>
</tr>
</tbody>
</table>

**MC68HC11A8P1 Equates:**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGBASE</td>
<td>$1000</td>
</tr>
<tr>
<td>IRQ_VEC</td>
<td>$1EE</td>
</tr>
<tr>
<td>SPRS</td>
<td>$29</td>
</tr>
<tr>
<td>SPIF</td>
<td>%00010000</td>
</tr>
<tr>
<td>SPCR</td>
<td>$28</td>
</tr>
<tr>
<td>SPDR</td>
<td>$2A</td>
</tr>
<tr>
<td>BAUD</td>
<td>$2B</td>
</tr>
<tr>
<td>SCCR1</td>
<td>$2C</td>
</tr>
<tr>
<td>SCCR2</td>
<td>$2D</td>
</tr>
<tr>
<td>TE</td>
<td>%00001000</td>
</tr>
<tr>
<td>SCR1</td>
<td>$2E</td>
</tr>
<tr>
<td>SCR2</td>
<td>$2F</td>
</tr>
<tr>
<td>PORTD</td>
<td>$8</td>
</tr>
<tr>
<td>SS</td>
<td>%00010000</td>
</tr>
<tr>
<td>DDRD</td>
<td>$9</td>
</tr>
</tbody>
</table>

**Misc. Equates:**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>$13</td>
</tr>
<tr>
<td>JMP</td>
<td>$7E</td>
</tr>
<tr>
<td>TOS</td>
<td>$35</td>
</tr>
<tr>
<td>BSCT</td>
<td></td>
</tr>
</tbody>
</table>

**Scratch Pad Storage Locations:**

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>%00010000</td>
</tr>
<tr>
<td>String</td>
<td>$8600</td>
</tr>
<tr>
<td>Reset</td>
<td>#TOS</td>
</tr>
<tr>
<td></td>
<td>#REGBASE</td>
</tr>
</tbody>
</table>

**Initialize the SCI Port**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA</td>
<td>%00010000</td>
</tr>
<tr>
<td>STAA</td>
<td>BAUD,X</td>
</tr>
<tr>
<td>BSET</td>
<td>SCCR2,X,#TE</td>
</tr>
</tbody>
</table>

9600 Baud (Crystal = 8.0 MHz)

Enable SCI's Transmitter

ASCII Carriage Return

"JMP $HHHH" Opcode

Top of Available RAM (BUFFALO Compatible)
Initialize the SPI Port

- `BSET DDRD,X,%00111000
- `LDA$ %01010100
- `STAA SPBR,X` Configure SS, SCK, and MOSI into Outputs
- Enable SPI, Bit Rate = 1 MHz, ...
- CPOL = 0, CPHA = 1

Initialize the Indirect Vector Table. The MC68HC11A8P1's interrupt vectors are in ROM, and can't be modified. They point to an indirect vector table placed in RAM, so users can set vector addresses at will.

- `LDA$ "JMP $HHHH" Opcode
- `STAA IRQ_VEC Store a "JMP IRQ" in the Table
- `LDD $IRQ Address of IRQ Interrupt Handler
- `STD $IRQ_VEC+1

The MC68HC68T1 is always reinitialized, because it needs to be set to line timebase. If the first-time-up bit is set in its status register, it must also have its time and date set.

- `BSET PORTD,X,#SS Enable the MC68HC68T1
- `LDA$ #STATUS Read the Status Register
- `JSR SQIURT Send Address to MC68HC68T1
- `JSR PORTD,X,#SS Dummy Access to get Returned Value
- `BCLR PORTD,X,#SS Disable the MC68HC68T1
- `TAB Save the Status to Check for First-Time-Up
- `BSET PORTD,X,#SS Enable the MC68HC68T1
- `LDA$ #CONTROL+WRITE Write the Control Register
- `JSR SQIURT Send Address

The MC68HC68T1 is set up for line timebase, but the crystal oscillator is set for 32.768 kHz operation, so it will properly run.

- `LDA$ #START+LINE+XTL_SEL1+XTL_SEL0 Select Line Operation
- `JSR SQIURT Control Register Value
- `BCLR PORTD,X,#SS Disable the MC68HC68T1

If this was the first time up for the MC68HC68T1, then set the time.

- `BITB #FIRST_UP Check First-Time-Up Bit
- `BEQ SKIP_SET Do Not Set the Time in the MC68HC68T1, if Clear
- `BCLR SET_CLOCK Set the Clock's Time Here

Enable the stop instruction.

- `Skip_Set TPA Get Condition Code Register Value
- `ANDA %01111111 Clear the Stop Mask
- `TAP
- `Main EQU
- `SEI
- `BSET PORTD,X,#SS Enter Critical Code Section
- `LDA$ #SECONDS Enable the MC68HC68T1
- `JSR SQIURT Read the Seconds Register First
- `JSR SQIURT Dummy Character to Shift in Seconds
- `JSR PORTD,X,#SS Save Seconds
- `JSR SQUIURT Read the Minutes Register
- `STAA TIME+2 Save Minutes
- `JSR SQUIURT Read the Hours Register
- `STAA TIME Save Hours
- `BCLR PORTD,X,#SS Disable the MC68HC68T1
- `CLI End of Critical Code Section

Convert B.C.D. time to ASCII.

- `LDA$ TIME Get Hours
- `BSR BCD_ASCI Convert to ASCII
- `STD STRING Build String
- `LDA$ TIME+1 Get Minutes
- `BSR BCD_ASCI Convert to ASCII
- `STD STRING+3 Save Room for Colon
- `LDA$ TIME+2 Get Seconds
- `BSR BCD_ASCI Convert to ASCII
- `STD STRING+6 Save Room for Colon
- `LDA$ # Colon
- `STAA STRING+2 Now Store the Colon
- `STAA STRING+5

562
Now send the time strong to the SCI port.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAB #8</td>
<td>8 Bytes to Send</td>
</tr>
<tr>
<td>LDY #STRING</td>
<td>Point to the String</td>
</tr>
<tr>
<td>SEND_CHR</td>
<td>Get the Character</td>
</tr>
<tr>
<td>LDA #Y</td>
<td>Output the Character</td>
</tr>
<tr>
<td>BSR OUTCH</td>
<td>Bump the Pointer</td>
</tr>
<tr>
<td>INY</td>
<td>Decrement Loop Counter</td>
</tr>
<tr>
<td>DEC B</td>
<td>Continue Until Loop Counter Exhaused</td>
</tr>
<tr>
<td>LDA #CR</td>
<td>Now Send a Carriage Return</td>
</tr>
<tr>
<td>BSR OUTCH</td>
<td>Continue in Main Loop</td>
</tr>
</tbody>
</table>

A power failure causes an IRQ to be generated by the MC34160 power supply chip. Enter here for power failure.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQ EQU</td>
<td>Interrupt Handler</td>
</tr>
<tr>
<td>LDX #REGBASE</td>
<td>Point to I/O Space</td>
</tr>
<tr>
<td>BSET PORTD.X.#SS</td>
<td>Enable the MC68HC68T1</td>
</tr>
<tr>
<td>LDA #CONTROL</td>
<td>Write the Control Register</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Select Crystal Operation</td>
</tr>
<tr>
<td>LDA #START!+XTL_SEL!+XTL_SEL0</td>
<td>Disable the MC68HC68T1</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Disable the MC68HC68T1</td>
</tr>
<tr>
<td>STOP</td>
<td>Just go to Sleep</td>
</tr>
</tbody>
</table>

Utility Subroutines:

This subroutine converts the packed BCD number in A into ASCII numbers in Register D.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCD_ASCII EQU *</td>
<td>Convert to ASCII</td>
</tr>
<tr>
<td>PSHA</td>
<td>Save Argument</td>
</tr>
<tr>
<td>ANDA #%00001111</td>
<td>Mask all but L.S.D.</td>
</tr>
<tr>
<td>ORAA #0</td>
<td>Convert to ASCII</td>
</tr>
<tr>
<td>TAB</td>
<td>Place in Low Half of Reg D</td>
</tr>
<tr>
<td>PULA</td>
<td>Restore Argument</td>
</tr>
<tr>
<td>LSRA</td>
<td>Align M.S.D.</td>
</tr>
<tr>
<td>LSRA</td>
<td></td>
</tr>
<tr>
<td>LSRA</td>
<td></td>
</tr>
<tr>
<td>ORAA #0</td>
<td>Convert to ASCII</td>
</tr>
<tr>
<td>RTS</td>
<td>Return with 2 ASCII Chars in Reg D</td>
</tr>
</tbody>
</table>

This is only a dummy time setting routine. The clock is set to 00:00:00 January 1, 1990, Friday (24 hour time). A real application would substitute an actual time setting routine here.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_CLOCK EQU *</td>
<td>Enable the MC68HC68T1</td>
</tr>
<tr>
<td>BSET PORTD.X.#SS</td>
<td>Write to the Seconds Register First</td>
</tr>
<tr>
<td>LDA #SECONDS</td>
<td>Write to the Seconds Register First</td>
</tr>
<tr>
<td>LDA #6</td>
<td>Write to the Hours Register</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Set Hours = 0</td>
</tr>
<tr>
<td>LDA #6</td>
<td>Write to the Minutes Register</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Set Minutes = 0</td>
</tr>
<tr>
<td>LDA #1</td>
<td>Write to the Seconds Register</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Set Seconds = 0</td>
</tr>
<tr>
<td>LDA #1</td>
<td>Write to the Date Register</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Set Date = 1</td>
</tr>
<tr>
<td>LDA #1</td>
<td>Write to the Month Register</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Set Month = January</td>
</tr>
<tr>
<td>LDA #1000</td>
<td>Write to the Year Register</td>
</tr>
<tr>
<td>BSR SQUIRT</td>
<td>Set Year = 1990</td>
</tr>
<tr>
<td>LDA #PORTD.X.#SS</td>
<td>Disable the MC68HC68T1</td>
</tr>
<tr>
<td>RTS</td>
<td>Return to Caller</td>
</tr>
</tbody>
</table>
Output the contents of Register A to the SPI Port. Return with A containing the character received from the MC68HC68T1. No other registers modified upon return. The slave select must be handled by the calling program.

```
SQUIRT    EQU
PSHX      #REGBASE
STAA      SPDR,X
WAITSPIF  BRCLR SPDR,X,#SPIF,WAITSPIF
          LDAA SPDR,X
          PULX
          RTS
```

Save Working Register
Point to I/O Registers
Send Byte
Loop Until Transmission Complete
Read Returned Data to Clear SPIF
Restore Working Register
Return to Calling Program

This subroutine transmits the contents of Register A to the SCI Port. All registers preserved upon return.

```
OUTCH     EQU
PSHX      #REGBASE
STAA      SCSR,X,#TDRE,WAITTDRE
          STAA SCDR,X
          PULX
          RTS
          END
```

Save Working Register
Point to I/O Space
Wait for Empty Transmit Buffer
Transmit Character
Restore Working Register
Return to Calling Program
NOTE: REMOVING THIS JUMPER CAUSES PROGRAM EXECUTION TO BEGIN AT $B600 (EEPROM START) RATHER THAN ENTERING BUFFALO, WHEN RESET IS NEGATED.

Figure 1. Schematic Diagram
Figure 2. Power Fail Timing
Interfacing the MC68HC05C5 SIOP to an I^2C Peripheral

By Naji Naufel

INTRODUCTION

When designing a system based on a standard, non-custom-designed, microcontroller unit (MCU), the user is faced with the problem of not having all the desired peripheral functions on-chip. This problem can be solved by interfacing the MCU to an off-chip set of peripherals. An ideal interface is a synchronous serial communication port. Unfortunately, these peripherals may not have a serial interface that is compatible with the Motorola simple synchronous serial I/O port (SIOP).

This document demonstrates how the SIOP on the MC68HC05C5 can be interfaced to an I^2C peripheral, the PCF8573 clock/timer. The MC68HC05C5 was chosen because its SIOP has a programmable clock polarity.

The serial peripheral interface (SPI) on the MC68HC05C4 cannot be used in the interface because the SPI pins cannot be used as output pins when the SPI is off.

SIOP DEFINITION

The SIOP (see Figure 1) is a three-wire master/slave system including serial clock (SCK), serial data input (SDI), and serial data output (SDO). A programmable option determines whether the SIOP handles data most significant bit (MSB) or least significant bit (LSB) first.

![Figure 1. SIOP Block Diagram](image-url)
SIOP SIGNAL FORMAT

The SCK, SDO, and SDI signals are discussed in the following paragraphs.

SCK

The state of SCK between transmissions can be either a logic one or a logic zero. The first falling edge of SCK signals the beginning of a transmission. At this time, the first bit of received data is presented to the SDI pin, and the first bit of transmitted data is presented at the SDO pin (see Figure 2). When CPOL = 0, the first falling edge occurs internal to the SIOP. Data is captured at the SDI pin on the rising edge of SCK. Subsequent falling edges shift the data and accept or present the next bit. When CPOL = 1, transmission ends at the eighth rising edge of SCK. When CPOL = 0, transmission ends at the eighth falling edge of SCK.

Format is the same for master mode and slave mode except that SCK is an internally generated output in master mode and an input in slave mode. The master mode transmission frequency is fixed at E/4.

SDO

The SDO pin becomes an output when the SIOP is enabled. The state of SDO always reflects the value of the first bit received on the previous transmission, if a transmission occurred. Prior to enabling the SIOP, PB5 can be initialized to determine the beginning state, if necessary. While the SIOP is enabled, PB5 cannot be used as a standard output since that pin is coupled to the last stage of the serial shift register. On the first falling edge of SCK, the first data bit to be shifted out is presented to the output pin.

SDI

The SDI pin becomes an input when the SIOP is enabled. New data may be presented to the SDI pin on the falling edge of SCK. Valid data must be present at least 100 ns before the rising edge of the clock and remain valid 100 ns after that edge.

SIOP REGISTERS

The SIOP contains the following registers: SCR, SSR, and SDR.
SIOP Control Register (SCR)

This register, located at address $000A$, contains three bits (see Figure 3).

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0A$</td>
<td>0</td>
<td>SPE</td>
<td>0</td>
<td>MSTR</td>
<td>CPOL</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RESET:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3. SIOP Control Register

SPE — Serial Peripheral Enable
When set, this bit enables the SIOP and initializes the port B data direction register (DDR) such that PB5 (SDO) is output, PB6 (SDI) is input, and PB7 (SCK) is an input in master mode. The port B DDR can be subsequently altered as the application requires, and the port B data register (except for PB5) can be manipulated as usual; however, these actions could affect the transmitted or received data. When SPE is cleared, port B reverts to standard parallel I/O without affecting the port B data register or DDR. SPE is readable and writable any time, but clearing SPE while a transmission is in progress will abort the transmission, reset the bit counter, and return port B to its normal I/O function. Reset clears this bit.

MSTR — Master Mode
When set, this bit configures the SIOP for master mode, which means that the transmission is initiated by a write to the data register and SCK becomes an output, providing a synchronous data clock at a fixed rate of the bus clock divided by 4. While the device is in master mode, SDO and SDI do not change function; these pins behave exactly as they would in slave mode. Reset clears this bit and configures the SIOP for slave operation. MSTR may be set at any time regardless of the state of SPE. Clearing MSTR will abort any transmission in progress.

CPOL — Clock Polarity
When CPOL is set, SCK idles high, and the first data bit is seen after the first falling edge. When CPOL is cleared, the SCK idles low, and the first data bit is seen after the first falling edge, which occurs internally (see Figure 2).

SIOP Status Register (SSR)
Located at address $000B$, the SSR contains only two bits (see Figure 4).

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0B$</td>
<td>SPIF</td>
<td>DCOL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RESET:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4. SIOP Status Register

SPIF — Serial Peripheral Interface Flag
This bit is set on the last rising clock edge, indicating that a data transfer has occurred. SPIF has no effect on further transmissions and can be ignored without problem. SPIE is cleared by reading the SSR with SPIF set, followed by a read or write of the SDR. If it is cleared before the last edge of the next byte, it will be set again. Reset also clears this bit.

DCOL — Data Collision
DCOL is a read-only status bit that indicates an invalid access to the data register has been made. A read or write of SDR during a transmission results in invalid data being transmitted or received. DCOL is cleared by reading the SSR with SPIF set, followed by a read or write of the SDR. If the last part of the clearing sequence is done after another transmission has been started, DCOL will be set again. If DCOL is set and SPIF is not set, clearing DCOL requires turning the SIOP off, then turning it back on using the SPE bit in the SCR. Reset also clears this bit.
SIOP Data Register (SDR)

Located at address $00OC$, SDR is both the transmit and receive data register (see Figure 5). This system is not double buffered; thus, any write to this register destroys the previous contents. The SDR can be read at any time, but if a transmission is in progress, the results may be ambiguous. Writes to the SDR while a transmission is in progress can cause invalid data to be transmitted and/or received. This register can be read and written only when the SIOP is enabled (SPE = 1).

<table>
<thead>
<tr>
<th>$0C$</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

Figure 5. SIOP Data Register

I$^2$C DEFINITION

The inter IC (I$^2$C) is a two-wire half-duplex serial interface with data transmitted/received MSB first. The two wires are a serial data line (SDA) and a serial clock line (SCL).

The protocol consists of a start condition, slave address, n bytes of data, and a stop condition (see Figure 6). Each byte is followed by an acknowledge bit. A start condition is defined as a high-to-low transition on SDA while SCL is high; a stop condition is defined as a low-to-high transition on SDA while SCL is high (see Figure 9). An acknowledge is a low logic level sent by the addressed receiving device during the ninth clock period. A master receiver signals the end of data by not generating an acknowledge after the last byte has left the slave device.

Figure 6. PCF8573 Serial Data Format

INTERFACING THE SIOP TO THE PCF8573

The PCF8573 has an address of 11010 A1 A0, where A1 and A0 give the device a one-of-four address assigned by two hardware pins. Bit 0 of the address byte is the read/write indicator (see Figure 7).

<table>
<thead>
<tr>
<th>$7$</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A1</td>
<td>A0</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Figure 7. Address Byte

The byte following the address byte is the mode pointer used to control register access inside the PCF8573. Subsequent bytes following the mode pointer contain data read from or written to the clock/timer. Clock data is in binary-coded decimal format with two digits per byte.
HARDWARE DESCRIPTION

The SIOP is used as master by setting the MSTR bit in the SPCR. PB7/SCK is connected to SCL. Since the PCF 8573 has a bidirectional data line (SDA) and the SIOP has separate input and output pins, the SDO and the SDI pins need to be connected. A resistor must be used for this connection because port B is not open-drain (see Figure 8). The SEC pin, which goes high every second, is connected to PA7, which is polled by software to keep a seconds count.

When receiving data from the clock timer, an $FF is transmitted by the SIOP, which makes the resistor (R3), in series with the SDO pin, look like a pullup to VDD; therefore, SDO will not interfere with data coming from the SDA pin.

Figure 8. MC68HC05C5 Connection to PCF8573
SOFTWARE DESCRIPTION

To generate the timing required by the I²C, the user has to manipulate the port B pins as I/O and SIOP pins (see Figure 9). Before any data transactions, PB5 and PB7 are initialized high. While the SIOP is off (SPE = 0), PB5 is cleared to zero while PB7 is still high, creating a start condition. The SIOP is then enabled with CPOLE = 0 and MSTR = 1 and a byte is transmitted. After transmission is complete, the SIOP is turned off (SPE = 0), and PB7 is toggled high, then low, to generate the acknowledge clock. If the MCU is sending data, PB5 is forced high during the acknowledge pulse; otherwise, it is forced low to let the slave know that the byte has been received. If needed, the stop condition is accomplished by clearing PB5, setting PB7, then setting PB5.

To satisfy the 100-kHz serial clock maximum rating of the PCF8573, the MC68HC05C5 must be slowed to run at a bus speed of 250 kHz, which gives a serial clock rate of 62.5 kHz.

Figure 9. SIOP-Generated Timing
SOFTWARE APPLICATION

In demonstrating how the SIOP is interfaced to an \text{i}^2\text{C} peripheral, the author developed a complex application having time and calendar functions.

This application interfaces serially with a terminal to allow the user to initialize the PCF8573 with the time and date (see Figure 8). After the software prompts the user to enter the date (month, day, hour, and minutes), it starts displaying the information every second (see Figure 10). Every second the SEC pin goes high, telling the software to read the PCF8573 data and display it along with the software-kept seconds.

To initialize clock data, use the following sequence:
- Send $D0$ with start bit (ADDRESS)
- Send $00$ without start bit (CONTROL)
- Send hours data without start bit
- Send minutes data without start bit
- Send day data without start bit
- Send month data without start bit
- Generate stop bit

To read clock data, use the following sequence:
- Send $D0$ with start bit (ADDRESS)
- Send $00$ without start bit (CONTROL)
- Set up for low acknowledge bit transmit
- Send $D1$ with start bit (ADDRESS)
- Send $FF$ without start bit to receive hours
- Send $FF$ without start bit to receive minutes
- Send $FF$ without start bit to receive day
- Send $FF$ without start bit to receive month
- Generate stop bit

Since the MC68HC05C5 does not have a universal asynchronous receiver transmitter (UART), the interface to the terminal was implemented in software. See subroutines INCHAR and OUTCHAR in APPENDIX A PROGRAM LISTING.
Figure 10. Program Flowchart
APPENDIX A. PROGRAM LISTING

0001  *********************************************
0002  "This program is written to demonstrate interfacing the MOTROLA
0003  * Simple Serial I/O (SIOP) bus to the SIGNETICS IIC bus.
0004  * The 2 devices used are the MC68HC05C5 MCU and the PCF8573 clock/timer.
0005  * Bus speed is 250 KHz.
0006  * The MCU is used as a master and the clock/timer is used as a
0007  * slave. Some software intervention has to be done so that the
0008  * SIOP meets all IIC specifications.
0009  * The MCU displays clock data on a terminal screen at 2400 baud
0010  *
0011  *
0012  * Written by: Naji Naufel
0013  * CSIC MCU Design
0014  *
0015  *
0016  *
0017  ********************************************************
0018  porta equ $00 ;port a data register
0019  portb equ $01 ;port b data register
0020  portc equ $02 ;port c data register
0021  ddra equ $04 ;port a data direction register
0022  ddb equ $05 ;port b data direction register
0023  ddrc equ $06 ;port c data direction register
0024  spcer equ $0a ;spi control register
0025  spdr equ $0b ;spi status register
0026  spdc equ $0c ;spi data register
0027  *
0028  raddr equ $d1 ;peripheral address for read
0029  waddr equ $d0 ;peripheral address for write
0030  *
0031  ram equ $80 ;start of ram space
0032  *
0033  org ram
0034  sec rmb 1 ;seconds byte
0035  control rmb 1 ;control byte
0036  ack rmb 1 ;acknowledge polarity
0037  hour rmb 1
0038  min rmb 1
0039  month rmb 1
0040  day rmb 1
0041  savx rmb 1
0042  savb rmb 1
0043  xtemp rmb 1
0044  count rmb 1
0045  InChar rmb 1
0046  OutChar rmb 1
0047  atemp rmb 1
0048  odelay rmb 1 ;delay variable for serial routines

575
licc5
0050
0051
0052
0053
0054
0055 0200
org $0200
0056
* all timing is based on a 500 Khz crystal
0057
0058
0059 0200
begin equ *
0060 0200 a6 02
lda $100000010
0061 0202 b7 00
sta porta ; TX pin high
0062 0204 b7 04
sta ddrca ; PA1=TX pin, PA0=RX pin
0063 0206 a6 a0
lda $110100000 ; pb7=pb5=output, pb6=input
0064 0208 b7 01
sta portb
0065 020a b7 05
sta ddrb
0066 020c ae 03
ldx $3
0067 020e d6 04 07
lda delays, x
0068 0211 b7 8e
sta cdelay ; 2400 baud
0069 0213 a6 10
lda $100010000 ; mstr=1, cpol=0, siop still off
0070 0215 b7 0a
sta spcr
0071 0217 b7 82
sta ack ;ack flag non-zero, high acknowledge
0072
0073 0219 cd 03 85
jsr crlf
0074 021c cd 03 76
jsr outmsg ;print "ENTER MM/DD HH:MM"
0075 021f cd 02 85
jsr getdate ;get month
0076 0222 b7 85
sta month
0077 0224 cd 03 90
jsr inchar ; dummy char ';
0078 0227 cd 02 85
jsr getdate ;get day
0079 022a b7 86
sta day
0080 022c cd 03 90
jsr inchar ; dummy space
0081 022f cd 02 85
jsr getdate ;get hours
0082 0232 b7 83
sta hour
0083 0234 cd 03 90
jsr inchar ; dummy substring
0084 0237 cd 02 85
jsr getdate ;get minutes
0085 023a b7 84
sta min
0086 023c cd 03 90
again jsr inchar ; wait for <CR>
0087 023f a1 0d
cmp #$0d
0088 0241 26 f9
bne again
0089 0243 cd 03 85
jsr crlf
0090
0091
* issue a reset prescaler command
0092
0093 0246 a6 20
lda #$20
0094 0248 b7 81
sta control
0095 024a cd 02 d2
jsr addrcntl
0096 024d cd 02 cb
jsr stop
0097
0098
* initialize the clock
0099
0100 0250 a6 00
lda #$00
0101 0252 b7 81
sta control
0102 0254 cd 02 d2
jsr addrcntl ; send address/control bytes
0103 0257 cd 02 db
jsr senddata ; send 4 data bytes
0104
0105
* issue a reset prescaler command
0106
0107 025a a6 20
lda #$20
0108 025c b7 81
sta control
0109 025e cd 02 d2
jsr addrcntl
0110 0261 cd 02 cb
jsr stop
0111
0112 0264 3f 80
clr sec ; clear seconds counter
0113 0266 0f 00 fd
sec_pin broclr 7, porta,* ; wait for SEC pin to go high
0114 0269 cd 03 31
jsr dispdata ; display clock data
0115 026c 0e 00 fd
brset 7, porta,* ; wait until pin goes low
0116 026f 20 f5
bra sec_pin
0117
0118 0271 45 4e 54 45 52 20 4d 4d
crac "ENTER MM/DD HH:MM"
0119 0282 0d 0a 04
fcb $0d, $0a, $04
0120
576
This routine reads 2 ASCII characters and converts them into 2 BCD digits in Acc. A.

*********************************************************************************

getdate equ *
  jsr    inchar   ;get character
  sub    #$30    ;convert to binary
  lsla
  lsla
  lsla
  lsla
  :make it upper nibble
  lsla
  jsr    inchar   ;get second ASCII char.
  add    save     ;2 BCD digit is in Acc. A now
  rts

*********************************************************************************

Convert a binary byte in Acc. A into a 2 digit BCD number in Acc. A and display it as 2 ASCII chars.

*********************************************************************************

bin_dec equ *
  clrx        ;clear number of subtraction counter
  sub_more    sub    #$10    ;see how many times it is divisible
  bmi         no_tens  ;by 10
  incx        ;increment counter
  bra         sub_more  ;subtract more tens
  no_tens     add    #$10    ;restore number to positive
  lsix        ;put 10's digit in upper nibble of X
  lsix
  lsix
  lsix
  lsix
  stx    save     ;merge both nibbles in Acc. A
  add    save     ;display the 2 digits
  rts
This subroutine transfers a byte from the HC05's SPI to the I2C.

* peripheral data is in reg X upon entry.

* _w_start_ is the entry point for sending a start bit.

* _start_ is the entry point for transferring data without a start condition.

---

```assembly
0172 02ad
0173 02ad 1e 01     bset 7, portb  ; take SCL line high
0174 02af 1b 01     bclr 5, portb  ; start condition
0175 02b1
0176 02b1 1c 0a     bset 6, spcr   ; enable SPI, SPE=1
0177 02b3 bf 0c     stx spdr   ; send data
0178 02b5 0f 0b fd   wait brcir 7, spcr, wait  ; wait for end of transmission
0179
0180 02ba 1d 0a
0181
0182 02ba 3d 82     tst ack   ; test acknowledge flag
0183 02bc 26 04     bne hi_ack  ; keep ACK high
0184 02be 1b 01     lo_ack bclr 5, portb  ; else, clear ACK bit
0185 02c0 20 02     bra hi_ack1 ; generate ACK clock
0186
0187 02c2 1a 01     hi_ack bset 5, portb  ; send high ACK bit
0188 02c4 1a 01     hi_ack1 bset 7, portb  ; take pb7 (SCL) high
0189 02c6 1f 01     bclr 7, portb
0190 02c8 1a 01     bset 5, portb  ; return data pin high
0191 02ca 81     rts
```
* the following routine (stop) creates a stop condition

```
stop equ *
```

```
bclr 5, porth ; bring sda low
bset 7, porth ; bring scl high
bset 5, porth ; bring sda high
```

* the following routine sends 2 bytes, an address byte followed by a control byte in control.

```
** the following routine sends 4 bytes. **
```

```
addrctl equ *
```

```
ldx #waddr ; (r/w=0)
```

```
lrx w_start ; send address with start condition
```

```
ldx control
```

```
ldx control ; send control byte without start
```

```
ldx nostart ; send hours
```

```
ldx min
```

```
ldx day
```

```
ldx month
```

```
ldx stop ; stop condition
```
* Output 2 BCD digits in A as ASCII chars.

```assembly
                         bcd    equ    sta  save
            :save A
3 0246 0247 0248 0249 0250
            jsr  outlhf :output left half
1 0251 0252 0253 0254 0255
            lda  save
3 0256 0257 0258 0259 0260
            jsr  outrhf :output right half
3 0261 0262 0263 0264 0265
                         rts
0266 0267 0268 0269 0270
                         lda  $500
1 0271 0272 0273 0274 0275
                         sta  control
3 0276 0277 0278 0279 0280
                         inca
1 0281 0282 0283 0284 0285
                         sta  ack :high ack bit (ack non-zero)
3 0286 0287 0288 0289 0290
                         jsr  addrctl ;send address/control
3 0291 0292 0293 0294 0295
                         lda  #raddr ;(r=1)
1 0296 0297 0298 0299 0300
                         bsr  w_start ;send address with start condition
3 0301 0302 0303 0304 0305
                         clr  ack ;low ack bit
3 0306 0307 0308 0309 0310
                         lda  #$ff ;and read 4 data bytes
1 0311 0312 0313 0314 0315
                         bsr  nostart ;keep mosi open drain (high)
3 0316 0317 0318 0319 0320
                         lda  spdr ;get received data
3 0321 0322 0323 0324 0325
                         sta  hour ;hours
3 0326 0327 0328 0329 0330
                         lda  $ff
1 0331 0332 0333 0334 0335
                         bsr  nostart
3 0336 0337 0338 0339 0340
                         lda  spdr
3 0341 0342 0343 0344 0345
                         sta  min ;minutes
3 0346 0347 0348 0349 0350
                         lda  $ff
1 0351 0352 0353 0354 0355
                         jsr  nostart
3 0356 0357 0358 0359 0360
                         lda  spdr
3 0361 0362 0363 0364 0365
                         sta  day ;days
3 0366 0367 0368 0369 0370
                         lda  $ff
1 0371 0372 0373 0374 0375
                         jsr  nostart
3 0376 0377 0378 0379 0380
                         lda  spdr
3 0381 0382 0383 0384 0385
                         sta  month ;months
3 0386 0387 0388 0389 0390
                         lda  spdr
3 0391 0392 0393 0394 0395
                         sta  b7 $85
3 0396 0397 0398 0399 0400
                         lda  spdr
3 0401 0402 0403 0404 0405
                         rts
3 0406 0407 0408 0409 0410
```
*******************************************************************
* This service routine increments the seconds counter
* and displays the clock data on the screen every second.
*******************************************************************

0300 0331 dispdata equ *
0301 0331 a6 0d lda #$0d
0302 0333 cd 03 ca jsr_putchar ;send <CR>
0303 0336 cd 02 f9 jsr_read ;read 4 bytes from clock
0304
0305 0339 b6 85 lda month ;display month
0306 033b a4 1f and #$1f
0307 033d ad af bsr bcd ;output 2 BCD digit
0308 033f a6 2f lda '/'
0309 0341 cd 03 ca jsr_putchar ;output '/'
0310
0311 0344 b6 86 lda day ;display day
0312 0346 a4 3f and #$3f
0313 0348 ad a4 bsr bcd
0314 034a a6 20 lda #$20
0315 034c cd 03 ca jsr_putchar ;output space
0316
0317 034f b6 83 lda hour ;display hours
0318 0351 a4 3f and #$3f
0319 0353 ad 99 bsr bcd
0320 0355 a6 3a lda '#'
0321 0357 cd 03 ca jsr_putchar ;output ':'
0322
0323 035a b6 84 lda min ;display minutes
0324 035c a4 7f and #$7f
0325 035e cd 02 ee jsr bcd
0326 0361 a6 3a lda ':'
0327 0363 cd 03 ca jsr_putchar ;output ':'
0328
0329 0366 b6 80 lda sec ;display seconds
0330 0368 cd 02 98 jsr_bin_dec ;convert seconds to BCD and display
0331
0333 036b b6 80 lda sec ;read seconds byte
0334 036d 4c inca ;increment it
0335 036e a1 3c cmp #60
0336 0370 26 01 bne not_sixty ;not 60 yet
0337 0372 4f clra
0338 0373 b7 80 not_sixty sec ;update seconds counter
0339 0375 81 rts

Page 7
The following are the various routines associated with displaying data.

```
; The print character string
outmag equ * ;print character string
clr x
prtmag lda mag,x ;get message character
cmp #$04 ;NOT yet?
beq finish ;Yes.
jsr outchar ;output character
inc x
bra prtmag
finish rts
```

Register A and location InChar receive the character typed, parity stripped and mapped to upper case. X is unchanged.
For HCS05C5 PA1 and PA0 are txd and rxd respectively.
* i.e. transmit from PA1 and receive from PA0
* Interrupts are masked on entry and unmasked on exit.

```
inchar equ * ;input character from terminal
stx xtemp ;save X
lda #$0 ;number of bits to read
sta count
getc4 nop ;unmask to allow service, then
sei ;mask while looking for start bit
brs et 0,.porta,* ;wait for hilo transition
lda #$2
brs r delay ;delay 1/2 bit to middle of start bit
brs 0, porta,getc4 ;false start bit test
```
outchar equ * ; output character to terminal
putc1 equ * ; sneaky entry from getc to avoid clobbering x
sta OutChar ; save it in both places
sta temp ; going to put out
sta count ; 9 bits this time
clrx ; for very obscure reasons
clex ; this is the start bit
sei ; mask interrupts while sending
bra putc2 ; jump in the middle of things

* main loop for outchar

putc5 ror OutChar ; get next bit from memory
putc2 bcc putc3 ; now set or clear port bit
best 1,ports ;5
bra putc4 ;3
clear 1,ports ;5
clear putc4 ;3
bra putc4 ;3
clear delay ;6
ldd #6
bdc delay ;6
dec count ;5
bne putc3 ;3 still more bits
best 1,ports ;5 send stop bit
bra putc2 ;5
temp
nop ;2 re-enable interrupts

* caller loop overhead assumes 24 cycles in external loop

delay equ *

* 1/2 bit delay = 24 cycles overhead + (6*A)+8, where A=2
* 1 bit delay = 24 cycles overhead + [(6*A)+8]+[(6*B)+8]+[(6*8)+1]+8, A=2, B=6

* delays for baud rate calculation

* delays fcb 32 ; 300 baud
fcb 8 ; 1200 baud
fcb 2 ; 4800 baud
fcb 1 ; 9600 baud
* Output the left nibble of Acc A as ASCII character.

```
outlfh     equ   *
lsla       
lsla       
lsla       
lsla       
outlhf     and   $$0f
add        $$30  ;make ASCII
jer        outchar ;send character to terminal
rts
```

```
irqv       fdb   begin
swiv       fdb   begin
resetv     fdb   begin
```
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>iicc5</td>
<td>00Bb</td>
</tr>
<tr>
<td>iicc5</td>
<td>008c</td>
</tr>
<tr>
<td>iicc5</td>
<td>0082</td>
</tr>
<tr>
<td>iicc5</td>
<td>02d2</td>
</tr>
<tr>
<td>iicc5</td>
<td>023c</td>
</tr>
<tr>
<td>iicc5</td>
<td>008d</td>
</tr>
<tr>
<td>iicc5</td>
<td>02ee</td>
</tr>
<tr>
<td>iicc5</td>
<td>0200</td>
</tr>
<tr>
<td>iicc5</td>
<td>0298</td>
</tr>
<tr>
<td>iicc5</td>
<td>008e</td>
</tr>
<tr>
<td>iicc5</td>
<td>0088</td>
</tr>
<tr>
<td>iicc5</td>
<td>0aek</td>
</tr>
<tr>
<td>iicc5</td>
<td>0082</td>
</tr>
<tr>
<td>iicc5</td>
<td>02d2</td>
</tr>
<tr>
<td>iicc5</td>
<td>023c</td>
</tr>
<tr>
<td>iicc5</td>
<td>0086</td>
</tr>
<tr>
<td>iicc5</td>
<td>0004</td>
</tr>
<tr>
<td>iicc5</td>
<td>0006</td>
</tr>
<tr>
<td>iicc5</td>
<td>0402</td>
</tr>
<tr>
<td>iicc5</td>
<td>0407</td>
</tr>
<tr>
<td>iicc5</td>
<td>0331</td>
</tr>
<tr>
<td>iicc5</td>
<td>0384</td>
</tr>
<tr>
<td>iicc5</td>
<td>0396</td>
</tr>
<tr>
<td>iicc5</td>
<td>03a2</td>
</tr>
<tr>
<td>iicc5</td>
<td>0285</td>
</tr>
<tr>
<td>iicc5</td>
<td>02e2</td>
</tr>
<tr>
<td>iicc5</td>
<td>0083</td>
</tr>
<tr>
<td>iicc5</td>
<td>0390</td>
</tr>
<tr>
<td>iicc5</td>
<td>0085</td>
</tr>
<tr>
<td>iicc5</td>
<td>0088</td>
</tr>
<tr>
<td>iicc5</td>
<td>0005</td>
</tr>
<tr>
<td>iicc5</td>
<td>0200</td>
</tr>
<tr>
<td>iicc5</td>
<td>02f9</td>
</tr>
<tr>
<td>iicc5</td>
<td>0271</td>
</tr>
<tr>
<td>iicc5</td>
<td>008a</td>
</tr>
<tr>
<td>iicc5</td>
<td>0390</td>
</tr>
<tr>
<td>iicc5</td>
<td>0390</td>
</tr>
<tr>
<td>iicc5</td>
<td>0376</td>
</tr>
<tr>
<td>iicc5</td>
<td>0271</td>
</tr>
<tr>
<td>iicc5</td>
<td>008d</td>
</tr>
<tr>
<td>iicc5</td>
<td>0088</td>
</tr>
<tr>
<td>iicc5</td>
<td>0085</td>
</tr>
<tr>
<td>iicc5</td>
<td>0390</td>
</tr>
<tr>
<td>iicc5</td>
<td>0376</td>
</tr>
<tr>
<td>iicc5</td>
<td>0271</td>
</tr>
<tr>
<td>iicc5</td>
<td>0088</td>
</tr>
<tr>
<td>iicc5</td>
<td>0085</td>
</tr>
<tr>
<td>iicc5</td>
<td>008d</td>
</tr>
<tr>
<td>iicc5</td>
<td>0088</td>
</tr>
<tr>
<td>iicc5</td>
<td>0005</td>
</tr>
<tr>
<td>iicc5</td>
<td>0200</td>
</tr>
<tr>
<td>iicc5</td>
<td>02f9</td>
</tr>
<tr>
<td>iicc5</td>
<td>0002</td>
</tr>
<tr>
<td>iicc5</td>
<td>0377</td>
</tr>
<tr>
<td>iicc5</td>
<td>03cc</td>
</tr>
<tr>
<td>iicc5</td>
<td>03db</td>
</tr>
<tr>
<td>iicc5</td>
<td>0361</td>
</tr>
<tr>
<td>iicc5</td>
<td>0365</td>
</tr>
<tr>
<td>iicc5</td>
<td>03d9</td>
</tr>
<tr>
<td>iicc5</td>
<td>00d1</td>
</tr>
<tr>
<td>iicc5</td>
<td>0080</td>
</tr>
<tr>
<td>iicc5</td>
<td>0269</td>
</tr>
<tr>
<td>iicc5</td>
<td>01ff</td>
</tr>
<tr>
<td>iicc5</td>
<td>0316</td>
</tr>
<tr>
<td>iicc5</td>
<td>0070</td>
</tr>
<tr>
<td>iicc5</td>
<td>0429</td>
</tr>
<tr>
<td>iicc5</td>
<td>0432</td>
</tr>
<tr>
<td>iicc5</td>
<td>0434</td>
</tr>
<tr>
<td>iicc5</td>
<td>0428</td>
</tr>
<tr>
<td>iicc5</td>
<td>0028</td>
</tr>
<tr>
<td>iicc5</td>
<td>0031</td>
</tr>
<tr>
<td>iicc5</td>
<td>0031</td>
</tr>
<tr>
<td>iicc5</td>
<td>0042</td>
</tr>
<tr>
<td>iicc5</td>
<td>0041</td>
</tr>
<tr>
<td>iicc5</td>
<td>0034</td>
</tr>
<tr>
<td>iicc5</td>
<td>0113</td>
</tr>
<tr>
<td>iicc5</td>
<td>0234</td>
</tr>
<tr>
<td>iicc5</td>
<td>0024</td>
</tr>
<tr>
<td>iicc5</td>
<td>000c</td>
</tr>
<tr>
<td>iicc5</td>
<td>000b</td>
</tr>
<tr>
<td>iicc5</td>
<td>02cb</td>
</tr>
<tr>
<td>iicc5</td>
<td>000a</td>
</tr>
<tr>
<td>iicc5</td>
<td>000c</td>
</tr>
<tr>
<td>iicc5</td>
<td>000b</td>
</tr>
<tr>
<td>iicc5</td>
<td>02cb</td>
</tr>
<tr>
<td>iicc5</td>
<td>0299</td>
</tr>
<tr>
<td>iicc5</td>
<td>01ff</td>
</tr>
<tr>
<td>iicc5</td>
<td>02ad</td>
</tr>
<tr>
<td>iicc5</td>
<td>00dc</td>
</tr>
<tr>
<td>iicc5</td>
<td>02b5</td>
</tr>
<tr>
<td>iicc5</td>
<td>0089</td>
</tr>
</tbody>
</table>

585
Pulse Generation and Detection with Microcontroller Units

By Mike Pauwels

INTRODUCTION

This application note examines two common interfaces between microcontroller units (MCUs) and external circuitry — pulse generation and pulse detection. Several families of Motorola MCUs and a variety of pulse applications are considered. Code segments and listings are also included.

PULSE GENERATION

MCUs are often required to generate timed output pulses — i.e., signals asserted for a specified period of time. The application can be strobing a display latch, transmitting a code, or metering a reagent in a process control system; however, each application has specific requirements for pulse duration and accuracy. This application note examines methods of generating these pulses in relationship to timing accuracy, coding efficiency, and other controller requirements.

The following paragraphs describe the timing of the signals — the start time and duration of the pulse. All pulses can be divided into three basic classifications: short pulses, long pulses, and easy pulses. Each class of pulses is considered using three MCUs with different timer structures.

On the low end of the scale is the MC68HC05J1. The simple timer in this device limits the accuracy of short pulses and requires a larger amount of software investment to produce a given pulse. The second MCU, the MC68HC705C8 and similar devices, has a 16-bit timer that is somewhat more powerful and flexible than the MC68HC05J1 timer. Finally, the MC68HC11A8 offers additional features in the 16-bit timer system, as well as the possibility of producing multiple pulses simultaneously.

Because time measurements are being considered, the clock frequency for the MCU is significant. For this discussion, each MCU is assumed to be operating at a 2.0-MHz internal, implying a 4.0-MHz crystal for the MC68HC05 and an 8.0-MHz crystal for the MC68HC11A8. The maximum speeds of these devices is somewhat higher, but these are commonly used values. Of course, these MCUs can all be operated at much slower clock speeds also. All times should be scaled to the actual clock frequency.
Short Pulses

The classification of short pulses may vary according to the accuracy of the required pulse and the available MCU resources. In general, pulses of a few tens of microseconds and longer are relatively easy to produce. Below this broad limit, the methods used to generate short pulses may vary greatly according to the specific requirements. To produce a strobe pulse whose minimum required duration is in the order of magnitude of the clock period only requires writing a port bit high, then low in consecutive operations:

```
PULSE   BSET           BIT0, PORTA
BCLR    BIT0, PORTA
```

This produces a pulse duration of 2.5 \(\mu s\) duration in the MC68HC05, and 3.0 \(\mu s\) in the MC68HC11A8. The longer time in the MC68HC11A8 is a consequence of a longer BSET/BCLR instruction formation — three bytes versus two bytes in the MC68HC05. This is compensated for by the ability to set and clear multiple bits in one instruction. The MC68HC11A8, however, provides for a 0.5 \(\mu s\) minimum pulse by using the resources of two timer compare registers.

If the requirement for the pulse is longer than 2 \(\mu s\), the above pair of instructions can be separated by no operation (NOP) instructions or even by useful instructions to stretch to the desired pulse width. There are two problems with this option. First, padding the instructions with NOPs consumes MCU resources. If there is some task that the MCU can accomplish between the set and clear, this is not too serious. More difficult is the possible requirement that the pulse duration be run-time variable. The flexibility of the busy-wire pulse timing can be extended by adding a loop:

```
SETUP   LDA            DURATION
PULSE   BSET           BIT0, PORTA
LOOP    DECA
        BNE            LOOP
        BCLR           BIT0, PORTA
```

The duration of the pulse is:

\[
2.5 + 3.0 \times DURATION \mu s \text{ for the MC68HC05,}
\]
and
\[
3.0 + 2.5 \times DURATION \mu s \text{ for the MC68HC11A8}
\]

Of course the previous code could be padded with any number of NOPs at 1.0 \(\mu s\) or with branch nevers (BRNs) at 1.5 \(\mu s\) either inside or outside the timing loop for more precise values. However, the variable resolution is 2.5 or 3.0 \(\mu s\).

Note that in these cases the on-board timers were not used. In the case of these short pulses, the overhead of setting-up and reading the timers would be about as long as the pulse being driven. When the required pulse width is long enough to use the timer, easy pulses are produced.

Easy Pulses

To produce a 10 ms pulse with the MC68HC11A8 controller, force an output compare pin high and read the timer (in an uninterrupted sequence). Add the 10 ms to the timer value and store the result in the corresponding output compare register. Next write the corresponding output level (OLVL) bit to zero and enable the interrupt (if desired). The pulse completes automatically. Three questions arise: 1) What is the shortest pulse that can be produced in this manner? 2) What considerations must be made in the MC68HC705C8 timer which does not have a force register? 3) What is the equivalent procedure for the MC68HC05J1 timer?
In the MC68HC11A8, two output compare registers can be combined to drive a single output. The elapsed time between the two events can be as little as one clock time; 0.5 μs if the prescaler is one. The code is as follows:

<table>
<thead>
<tr>
<th>PULSE</th>
<th>LDD</th>
<th>TIMER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LDD</td>
<td>ADDD</td>
</tr>
<tr>
<td></td>
<td>$50</td>
<td></td>
</tr>
</tbody>
</table>

* The delay is selected according to
* timer prescaler, interrupts, etc.
* (min 33)

<table>
<thead>
<tr>
<th>STD</th>
<th>TOC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD</td>
<td>$1</td>
</tr>
</tbody>
</table>

Pulse Width

<table>
<thead>
<tr>
<th>STD</th>
<th>TOC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>$540</td>
</tr>
<tr>
<td>STAA</td>
<td>OC1M</td>
</tr>
<tr>
<td>STAA</td>
<td>OC1D</td>
</tr>
<tr>
<td>LDA</td>
<td>$580</td>
</tr>
<tr>
<td>STAA</td>
<td>TCTL1</td>
</tr>
</tbody>
</table>

Drive A6 Low

With the MC68HC705C8 timer system, there is no force bit for compare. The only way to drive the timer compare (TCMP) line high is to set the OLVL bit in the timer control register (TCR) and wait for a match. The exact start time of the pulse is easily obtained from the output compare register (OCR), so pulse accuracy is unaffected for moderate pulses. Often the pulse is started as soon as possible, if for no other reason than to complete the pulse setup routine. The following code segment provides a pulse start in 12 μs, assuming no interrupts.

* START THE PULSE

| BSET | OLVL, TCR |

* OUTPUT_COMPARE: = TIMER + DELAY

must be read first

<table>
<thead>
<tr>
<th>LDX</th>
<th>ACHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>ACLR</td>
</tr>
<tr>
<td>ADD</td>
<td>$DELAY</td>
</tr>
<tr>
<td>BCC</td>
<td>0C1</td>
</tr>
<tr>
<td>INX</td>
<td></td>
</tr>
</tbody>
</table>

mark time

<table>
<thead>
<tr>
<th>OC1</th>
<th>STX</th>
</tr>
</thead>
<tbody>
<tr>
<td>STA</td>
<td>OCHR</td>
</tr>
<tr>
<td>STA</td>
<td>OCLR</td>
</tr>
</tbody>
</table>

inhibits TOC

enables TOC

Using a value for DELAY of about 21 (cycles) results in an average latency of 12 μs after the beginning of this routine. Note that loads and stores to the 16-bit registers are always performed high-byte first to take advantage of special hardware that maintains coherency in 16-bit data transfers. The pulse will turn on 1 μs later when there is a carry out of the low-byte add, which should occur about 1 in 12 times.
The programming of a moderate length pulse is now quite trivial. Simply add the desired pulse width (at 2 μs per bit) to the value stored in the output compare. Write the new value to the OCR and set the OLVL bit to zero. To finish code segment:

```
*WAIT FOR PULSE TO BE SET
HERE
BRCLR OCF,TSR,HERE
LDA PW_L
ADD OCLR
TAX
LDA PW_H
ADC OCHR
STA OCHR
STX OCLR

*DONE!
```

The interrupt structure is not required to generate pulses. The 16-bit timers on the MC68HC11A8 and the MC68HC705C8 will automatically drive the falling edge of these pulses without software intervention. On the MC68HC05J1, there is no hardware timer interface. To drive moderate length pulses with this device, employ the interrupts so that useful work can be performed while the pulse is being timed. Consider a 10 ms pulse using the MC68HC05J1.

The simple timer of the MC68HC05J1 provides only the capability of being interrupted periodically. The source of interrupt can be a timer overflow or a real-time interrupt (RTI). The choice of interrupt times is given in Table 1:

<table>
<thead>
<tr>
<th>RTI/RT0</th>
<th>RTI</th>
<th>COP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>8.2 ms</td>
<td>57.3 ms</td>
</tr>
<tr>
<td>0 1</td>
<td>16.4 ms</td>
<td>114.7 ms</td>
</tr>
<tr>
<td>1 0</td>
<td>32.8 ms</td>
<td>229.4 ms</td>
</tr>
<tr>
<td>1 1</td>
<td>65.5 ms</td>
<td>458.8 ms</td>
</tr>
</tbody>
</table>

Consider the algorithm for the timing of a pulse as counting “ticks” on a clock. Initially, it seems the ticks of the timer must be counted — 5,000 ticks (2 μs per tick) for the desired period of 10 ms. However, the timer overflow and real-time interrupts of the MC68HC05J1 provide long ticks that sound their completion with interrupts. Instead of 5,000 short ticks, count as follows:

- 1 RTI tick of 8192 μs = 8,192
- 3 TOF ticks of 512 μs = 1,536
- 544 cycles of 0.5 μs = 272

**TOTAL** = 10,000 μs

= 10.000 ms
For most of this time background tasks can continue processing. The 544 cycles of busy-wait time include necessary work to set up the pulse. The key problem is the required timing of the start of the pulse. If the start time is flexible the design of the pulse could follow the pattern of Figure 1.

![Image](image1.png)

**Figure 1. Time Line for a 10 ms Pulse with Flexible Start Time**

Start the pulse on the next RTI service routine, then count timer overflow flags (TOFs) after the next RTI until the final sequence, which is timed by a busy-wait counter. Careful calculation of the latencies and instruction cycles produces a pulse with a high degree of accuracy.

When the start time is not as flexible, a different approach is necessary. Since it is now impossible to align the RTI boundaries with the pulse, use only the TOF ticks to time the pulse. To turn the pulse on as soon as possible, read the value of the timer at turn-on. Calculate the time until the next overflow, add the predicted turn-off execution time, and then determine how many full TOF periods are in the remainder. After subtracting these long ticks, the remaining value is the busy-wait. A time line for this approach is given in Figure 2.

![Image](image2.png)

**Figure 2. Time Line for a 10 ms Pulse with Immediate Start Time**
Since an interrupt occurs every 512 μs, the performance of the MCU degrades slightly — about 10% versus 1% for the first approach. The following code yields a 10.0 ms pulse on port A1, with a latency of 2.5 μs after the code is entered:

```
* ASSUME THE DESIRED PULSE WIDTH,
* RESOLVED TO 2 μs PER BIT,
* IS STORED IN A TWO-BYTE LOCATION
* LABELED: PW_H:PW_L FOR A
* PULSE WIDTH OF 10 ms THIS
* VALUE WOULD BE $1388

* TURN ON THE PULSE

START       BCLR     TOF, TCSR
            LDA      TIMER
            BSET     BIT0, PORTA
COMA        = TIME REMAINING
            SBA      PW_L
            BCC      PW1
            DEC      PW_H
            PW1
            LDX     #$3AA
            MUL     PW_L
            STX     PW_L
            BSET    TOFE, TCSR
            CLI

* THE TIMER INTERRUPT DOES THE REST
* OF THE WORK:
*
TOFI        DEC      PW_H
            BNE      END_T
            LDA      PW_L
            BEQ      PLS_L
LOOP        DECA
            BNE      LOOP
PLS_L       BCLR     BIT0, PORTA
            RTI
```

There will be some small inaccuracies due to latency of the interrupt and border conditions for the pulse width. The pulse can be refined, but if one-clock precision is required, choose another processor.

**Long Pulses**

The idea of using the interrupt structure to count long “ticks” can be expanded beyond one byte. If a two-byte decrement is performed in the previous MC68HC05J1 example, pulses up to 30 seconds in length can be generated. The inaccuracies are the same in absolute terms as for the shorter pulses; therefore the percent of error is much smaller.

The same approach is used to expand the pulse width that can be generated by the 16-bit timers in the MC68HC705C8 and the MC68HC11A8 processors. With the help of the output compare function, one-tick accuracy with very long pulses is possible. The accuracy of the output is determined only by the accuracy of the crystal. The code listed in Appendix A has been tested in an MC68HC05C4 and produced pulses in the order of one minute with an accuracy of one part per million. Code to generate
long pulses with an MC68HC11A8 is similar. Since the timer interrupts are used to count the ticks, most of the MCU resources are available to do background tasks. For example, the timer interrupt routine consumes less than 25 μs every 131 ms. This represents about 0.2% of the processing power of the MCU. The actual code takes about 200 bytes of memory. The pulse will be precise if the interrupts are not masked for more than about 130 ms at a time. Beyond that limit, whole ticks of 131 ms will be added.

Finally, the MC68HC11A8 timer system provides for two timer output functions to drive a single pin. With this MCU, the start time and end time of the pulse can be driven independently with differences of as little as one count between the two pulse edges.

**Summary — Pulse Generation**

Many MCU systems interface to hardware systems by means of timed pulses. Modern MCUs handle these pulses in three different ways depending on the hardware timing structure available and the length of the pulse.

Short pulses, ranging in length from as short as a microsecond to a few tens of microseconds, are usually timed with "busy-wait" loops. There is simply not enough time to set up a peripheral to control a pulse of short duration. The accuracy and resolution of these pulses is determined in part by the discrete execution time of branch instructions in the controller. The MC68HC11 can drive a pulse as short as a microsecond, resolved to a microsecond, by using the resources of two timer compare registers.

Moderate length pulses are simple to drive automatically using the 16-bit timer available in the MC68HC11A8 and many of the MC68HC05 MCUs. These are set-and-forget systems that run to completion typically in 131 ms. In the simpler MC68HC05 MCUs, there is no 16-bit timer, and the moderate length pulses must use the timer overflow interrupt to count out large chunks of the pulse time while some background task is being performed.

The approach is similar in the MCUs with the 16-bit timer when the desired pulse is greater than 131 ms. Multiple timer overflows can be counted in a few memory locations to produce very long pulses.

For more complex timing functions, a system may require a separate timing processor. In some complex control applications, an MC68HC11A8 or an MC68HC05 is employed as a peripheral timer to a larger computational engine. A variation on this theme is the time processing unit (TPU) in the MC68332. This complex timing system can perform several different functions on 16 different channels simultaneously, independent of the main processor. Information on the MC68332 is available from your Motorola Sales Office.

**PULSE DETECTION**

Another system problem encountered when applying an MCU to a physical system is the detection and measurement of pulses. These can range from the actuation of a pushbutton to pulse codes detection, detection of the period of rotation of an engine, and accumulation of the 'on' time of a process control valve. The periods can range from microseconds to minutes, hours, or more.

There are several parameters that characterize a pulse, as Figure 3 illustrates. As far as a digital system is concerned, most of these parameters cannot be measured directly by a digital device such as an MCU. Indeed, some parameters such as the voltage level must be modified before the pulse
is applied to the MCU. If the values of these parameters are interesting to the system, then an external
device such as an analog-to-digital converter is required. Other parameters may not be measurable
by the MCU, including the signal rise and fall times and the presence of noise on the signal.

![Figure 3. Characteristics of a Pulse](image)

Digital pulses convey information only in the timing of their signals, assuming that all voltage signals
vary between VSS and VDD, and that rise and fall times are sufficiently fast to be unambiguous to
the processor. The parameters of interest are the start-time and duration of the pulse. Noise, if it exists,
is interesting only to the extent that it can be seen by the controller, and in that case, provision must
be made to reduce its effects.

There remains one significant question to address before software design can commence. What is
the expected duration range of the pulse? There are no effective maximum limits on the duration
which can be measured; but very short pulses may require the support of on-chip or off-chip hardware.
A related characteristic is the start-time of the pulse, measured from some reference. This can be
thought of as the measurement of a pulse off-time, and hence is not significantly different from the
duration measurement. Also important is the required accuracy of the measurement, specified in
absolute or relative terms.

One important choice the designer makes in addressing system problems is the type of MCU that will
be used. Most MCUs have some sort of timing device on-chip. Within the five basic families of
Motorola processors are several timer variations. These range from the simple counter in the
MC68HC05J1 to the sophisticated TPU of the MC68332. The former is useful only for the simplest
requirements, while the latter can measure pulses accurately without intervention of the CPU. The
choice for most applications is usually between an MC68HC11 and one of the large family of
MC68HC05 devices.

The timer on the MC68HC11A8 provides as many as four hardware input signals with several
hardware registers to measure input events. By combining two input capture functions, or by using
the clock gate input of the MC68HC11A8, many pulse measurement problems are easily solved. It
is more difficult to address the problems with the 16-bit timer system found on the most popular MCU
family, the MC68HC05.
Consider the accuracy limitations of the MC68HC(7)05Cx 16-bit timer. The timer counter itself is incremented once for every eight cycles of the MCU crystal frequency. A 4.0-MHz crystal provides a count resolution of 2 μs. With short pulses, this resolution may be a contributor to accuracy limitations. For example, measuring a 50 μs pulse, this resolution will produce a count of 25 with a 1-bit quantizing error, an uncertainty of 4%. However, in measuring a one minute pulse, the quantizing error is 0.0000033%.

In the case of the longer pulses, the accuracy of the crystal can contribute far more to the loss of precision. A limited sampling of clock frequencies on MC68HC05 Evaluation Modules indicates that typical crystals may produce errors of 0.001%. While crystals can be selected or trimmed to much higher accuracy, it is important not to specify accuracies from the software that cannot be supported by the hardware.

Consider four general classes of pulses to detect: 1) very fast pulse, say 20 μs or less; 2) longer pulses up to 130 ms; 3) long pulses; and 4) noisy pulses. The second class is almost trivial with the TCAP feature of the MC68HC05. Indeed, these are the most common class of pulses, and the hardware does almost all of the work. These are considered a special case of the third class of pulses. The other three classes require a bit more study.

**Short Pulses**

To measure very fast pulses with the MC68HC05, it is necessary to deal with interrupt latency which can be as much as 10 μs. If an IRQ is triggered on the start of the pulse, the pulse may have ended by the time code is executed in the interrupt routine. Accuracy is limited by the latency of the system. An example of the code necessary to measure these pulses is given below:

```
INTERRUPT:
  CLRA
  BIL
  T_LOOP:
    INCA
    BIH
    T_LOOP
    COUNT LOOPS
A:

END_PULS:
  *
  *
```

After the pulse is driven low on the IRQ line, the timed wait is executed for the rising edge which enables detection of a very short pulse. At END_PULS, the Accumulator has a measurement of the length of the pulse resolved to 3 μs per bit. Assuming the interrupts are not masked the worst case time to get to point A the first time is 13.4 μs (11.5 μs if MUL is not used in the background). The fastest time is 9 μs. Any pulse shorter than this will result in a zero time value. If the pulse value is greater than zero, the pulse width is 3 μs times the accumulator value plus a latency time of 9 – 13.5 μs. Finally, the longest pulse time that this routine can reasonably measure before the accumulator will overflow is about 770 μs. The interpretation of the result is left to the user.

If a short pulse is brought in on the TCAP line, there is additional latency to consider. If there is sufficient time to reverse the IEDG bit and clear the ICF (minimum about 20 μs), this is a class 2 pulse. If the pulse is shorter than this, the input capture function may miss the second edge. Unlike the MC68HC11A8 input capture functions, the MC68HC05 timer pin (TCAP) is not directly detectable. Precautions must be taken in the hardware design if very short pulses are possible. For example, a port line could be wired to the TCAP pin and the state of the pin could be tested with a BRSET/BRCCLR. The minimum resolvable pulse length is still no better than the IRQ driven case. However, using the
TCAP input offers capability to measure pulses of either polarity up to 131 ms in length and with a resolution of 2 µs.

Of course, if the pulse is expected to be short and the start time can be predicted, a busy-wait can be executed for both ends of the pulse. In this case it is necessary to continually test the state of the input pin and branch accordingly. For example, if the expected pulse length is between 5 and 100 µs, execute a string of tests as shown below:

```
PULSE:

PO
BRCLR PIN, PORT, P0 WAIT FOR THE FIRST EDGE
BRSET PIN, PORT, P1 ACTUALLY USING THE CODE TO MEASURE
BRSET PIN, PORT, P2
.
.
.
BRSET PIN, PORT, PN

PN
INCA

.
.
.
.
.
.
.
.
.
.
.
.
.
P0
INCA
.
P1
INCA
*
* ACC CONTAINS PULSE WIDTH OF 2.5 µs
* PER BIT
```

This code yields a resolution of 2.5 µs for any pulse down to 2.5 µs. Below that, the pulse may be missed. As the expected pulse length gets larger, this code becomes unwieldy and finally impossible.

The addition of an instruction loop shortens the code at the expense of resolution:

```
PULSE:

PO
CLRA
BRCLR PIN, PORT, P0
BRSET PIN, PORT, P1
          * ACC CONTAINS PULSE WIDTH OF 4 µs PER BIT
```

For longer (class 2) pulses, use the input capture register of the timer to do all the work. Where the pulse is more than a few tens of microseconds long, the interrupt structure works well to measure the pulse within the accuracy of the crystal. The rising edge of the pulse triggers a first interrupt, and the service routine enables the interrupt on the falling edge. By reading the input capture register on each edge, the exact pulse length can be measured. This class of pulses is included in a special case of the long pulses below.

**Longer Pulses**

What if the pulse length exceeds the rollover time of the timer? By counting the rollovers, a pulse of arbitrary length can be measured. Consider the possibility of a 60 second pulse that must be detected
and measured accurately. If the timer counts 2 \( \mu s \) per bit, 30 million counts must be accumulated. To store this information, \( \log_2 (30,000,000,000) = 25 \) bits, or 4 bytes are needed. To be precise, a value of \( \$1 \text{ C9 C3 80} \) is expected.

The 16-bit timer will automatically record the edges of the pulse. Ignoring the overflow, if the start time is subtracted from the end time the result will yield the two least significant bytes of the pulse width. In the 60 second example, if the pulse is exactly correct, the difference between the output compare value at the start of the pulse and the value at the end of the pulse will be \( \$C380 \). Between those two pulse events, the timer will roll over \( \$1C9 \times 457 \). Counting those rollovers exactly will determine the pulse length. The timer overflow facility will allow a count of the rollovers under interrupt control. Some problems remain in arbitrating the interrupts and protecting for boundary conditions, which will be discussed below.

The general approach taken for the MC68HC05Cx TCAP works as well for the MC68HC11 family when a single input capture function is used for measurement.

Appendix B is a listing of an MC68HC05 program which can measure very long pulses with single tick accuracy. The program was tested with the pulse generation problem listed above and appears to work within the accuracy of the crystal. Some adjustment may be necessary when this software is integrated into the user's program, particularly insofar as the interrupt latency is affected, but the basic structure of the routines will perform the measurement function. Note that class 2 pulses can be measured with this routine as it stands, although some code savings can be realized if the pulse to be measured is known to be contained in less than 4 bytes.

Three particular areas should be attended to when incorporating this software in a larger project. The measurement routine uses mutually exclusive interrupts and no subroutines, therefore its contribution to stack push is seven bytes. Add this to any other subroutine and interrupt stack usage to determine the maximum stack depth and therefore the available RAM.

If other interrupts are used, remember that the interrupt mask is automatically set when the interrupt routine is entered. If the mask cannot be cleared, the execution time of the other interrupt, plus its latency, must be kept somewhat less than 500 \( \mu s \) (or the pulse width, whichever is smaller) to preserve the accuracy of the measurement routine. The same is true if critical code sections must be preserved with \( \text{SEI...CLI} \) instructions.

Within these limitations, the automatic timing features of the TCAP will provide accurate measurement of the pulse. The 500 \( \mu s \) limitation is necessary to assure the correct handling of the boundary conditions when an overflow coincides with a pulse edge. If the interrupts must be masked for longer periods, the boundary conditions handling can be modified.

The third area to consider is the effect of the interrupts on execution speed of the processor. The pulse measurement routines take less than 0.02% of the clock cycles when measuring long pulses, so the routine will not significantly affect the throughput of most programs, however, each timer overflow interrupt takes about 24 \( \mu s \), so software timing loops and critical sections can be affected.

**Noisy Pulses**

The important thing to remember about noisy pulses is that a noise edge often cannot be distinguished from a pulse edge. This is particularly true when the input capture register is used to detect the edge. But even when the edge is polled, a momentary change in the signal level can be erroneously interpreted.
In general, it is difficult to measure any true pulse that contains noise pulse durations in order of magnitude of the measurement resolution. This means that signals must be free of 1 μs noise pulses for most MCU detection and measurement algorithms. The MC68HC11A8 pulse accumulator function in gated mode can be used to measure the total asserted time of a very noisy pulse.

Often, the easiest way to eliminate the ambiguity of minor noise is with some low pass hardware filtering. Remember that low pass filtering will also round and delay the edges of the pulse. The delay will contribute more or less to the accuracy of the measurement. In addition, sampled edges can be double-checked in our busy-wait algorithms with the addition of a single instruction per edge:

```
PULSE:
  CLRA
  BRCLR PIN, PORT, P0
  BRCLR PIN, PORT, P0
  INCA
  BRSET PIN, PORT, P1
  BRSET PIN, PORT, P1

* * *

* ACC HAS PULSE AT 6.5 μS PER BIT
```

Sophisticated digital filter algorithms can be used to extract a pulse from very noisy conditions, but these are beyond the scope of this application note. Consider a simple method of determining the approximate pulse-width of an input signal corrupted by a lot of noise.

Consider the signal of Figure 4. Is this one noisy 5 ms pulse or a number of smaller pulses? Taken at face value, this would translate into a number of various length disjoint pulses. However, if this were part of a pulse-width modulated code that had been transmitted on an r.f. carrier, the range of reception of the pulse could be significantly improved if the intelligence could be unambiguously extracted from this waveform. Much of the success of decoding algorithms depends on the knowledge of the expected signal. If, for example, the above waveform is expected to be either a 6 ms pulse or a 2 ms pulse, it is expected that this algorithm would more often choose the former. If there were some independent cross check on the validity of the code detection, such as a cyclic redundancy check, the detection could be made with a fair degree of certainty.

![Figure 4. Noisy Pulse](image.png)

It is beyond the scope of this note to present a detailed discussion of pulse train encoding and decoding, but the following paragraphs offer a few ideas about developing an effective method for capturing noisy pulses with an MCU.
The detection of the above signal with any of the earlier methods is unlikely to yield the correct data. With the MC68HC11A8 pulse accumulator, the pulse can be determined to be more likely 6 ms than 2 ms, but without the pulse accumulator, the MC68HC05 MCUs require more software intensive methods.

The basic strategy used to measure the pulse is to take periodic samples of the signal and employ some heuristic process to discover the signal in the noise. Most commonly, the selected algorithm is simple voting. Additionally, some cross check of the data such as a check-sum may be employed. If, for example, a 100 µs sample of the above pulse is taken, marked by the tick marks on the drawing, the findings show that the signal is high for 37 to 50 samples. This is more consistent with a 'wide' pulse than a 'narrow' one. If a cross check agrees with this conclusion, there is some confidence in the conclusion. If the cross check disagrees, the error could be guessed based on the lowest probability detection; or a re-transmission might be required. If no cross check is possible, a decision can be made on a minimum probability required to accept the data.

Below is a sampling routine that uses the output compare interrupt to time the samples. Fifty samples are accumulated before testing the vote.

```
TOCI BRCUl TOF, TSR, NO_TOc CLEAR FLAG
LDX OCHR
LDA OCLR
ADD #50 INT IN 100 µs
BCC SMP1
INCX
SMP1 STX OCHR
STA OCLR
* NEXT SAMPLE IN 100 µs
*

BRCLR PIN, PORT, SMP2
INC VOTE
BRA SMP3
SMP2 DEC VOTE
DEC COUNT
BNE SMP9
*
* HERE AFTER 50 SAMPLES
* PUT VOTING ROUTINE HERE
*
NO_TOc:
*
* DO OTHER INTERRUPT HERE
*
RTI
```

Note that this interrupt routine consumes only about 25 – 30% of the processor cycles. This number is directly related to the sample rate — sampling of 1/2 the rate reduces usage to less than 15% of the processor.

The choice of voting algorithm is application dependent. However, synchronization of the signal must also be considered. Depending on the type of coding used, a signal can be assumed to be self-synced. That is, the measurement of any pulse after a quiet period causes the receiving processor to try to wake up to a wide pulse or a narrow pulse. This causes the voting algorithm to reject pulses that vary widely from one of the expected widths.
With crystal-controlled oscillators in both the transmitting processor and the receiving one, this does not present a problem. If one or both of the controller clocks is not tightly regulated, however, the receiver will require time base as well as start time synchronization. In general, the more information that must be transmitted, the greater potential for error due to noise. The information transmitted is the code, the start time, and the time base.

**Summary of Pulse Detection**

MCU systems often read information from a hardware device by means of timed pulses. When these pulses fall in the range of a few tens of milliseconds, most MCUs can measure the pulse width easily with a high degree of accuracy. When the pulses are very short, very long, or noisy, the accurate detection and measurement of them is more difficult.

The most important decision to be made in system design for pulse measurement is the choice of MCU, specifically the timer subsystem. The least sophisticated timers such as found on the MC68HC05J1 lose some resolution and accuracy, particularly for short pulses, but these simple timers are often found on the low-cost chips. As the complexity and cost of the timer is increased, so is the performance of the MCU in this task. The very complex timer system in the MC68332 provides the greatest resolution and performance of any MCU available. For information, call your local Motorola sales office.
APPENDIX A
TTL LONG PULSE GENERATION

TIC-TOC ROUTINES FOR 68HC05CX

WRITTEN 11/11/89 BY MIKE PAUWELS

PULSE GENERATION

THIS ROUTINE GENERATES PULSES FROM A MC68HC05CX MICROCONTROLLER USING
THE TIMER OUTPUT COMPARE FUNCTION. THE LENGTH OF PULSES GENERATED
RANGE FROM A FEW MICROSECONDS TO MORE THAN TWO HOURS.

THIS SOFTWARE IS INTENDED AS A SUBSYSTEM TO BE INCLUDED IN A LARGER
PROGRAM. ETC.

CONSTANTS:

SYSTEM CONSTANTS:

ADDRESSES:

OPT  nol
PORTA  EQU 0 PORT A
DDRA  EQU 4 DATA DIRECTION REGISTER FOR PORT A
TCR  EQU $12 TIMER CONTROL REGISTER
ICIE  EQU 7 INPUT CAPTURE INTERRUPT ENABLE
OCIE  EQU 6 OUTPUT COMPARE INTERRUPT ENABLE
TOIE  EQU 5 TIMER OVERFLOW INTERRUPT ENABLE
IEdg  EQU 1 INPUT EDGE
OLVL  EQU 0 OUTPUT LEVEL
TSR  EQU $13 TIMER STATUS REGISTER
ICF  EQU 7 INPUT CAPTURE FLAG
OCF  EQU 6 OUTPUT COMPARE FLAG
TOF  EQU 5 TIMER OVERFLOW FLAG
ICHR  EQU $14 INPUT CAPTURE REGISTER HIGH BYTE
ICLR  EQU $15 INPUT CAPTURE REGISTER LOW BYTE
OCHR  EQU $16 OUTPUT COMPARE REGISTER HIGH BYTE
OCLR  EQU $17 OUTPUT COMPARE REGISTER LOW BYTE
CHR  EQU $18 TIMER/COUNTER HIGH BYTE
CLR  EQU $19 TIMER/COUNTER LOW BYTE
AChr  EQU $1A ALTERNATE TIMER/COUNTER HIGH BYTE
ACLR  EQU $1B ALTERNATE TIMER/COUNTER LOW BYTE
OPT  1

PROGRAM CONSTANTS

ORG  $20

DELAY  FCB  6 DELAY FOR START OF PULSE
MIN_PLS  FCB  5 MINIMUM PULSE WIDTH IN CLOCK COUNTS
DO_PLS  FCB  $01,$C9,$C3,$80

VARIABLES

ORG  $8A OR CONCATENATE WITH USER MEMORY
PULSE  RMB  4 MAX TIME = 143.1655765 MINUTES!
ASSUMING A 4 MHZ CRYSTAL, FOUR BYTES WILL AUTOMATICALLY TIME
2^33 MICROSECONDS (ABOUT 2.4 HOURS) WITHIN THE ACCURACY OF THE
CRYSTAL. EACH BIT IS 2 MICROSECONDS. FOR LONG TIME PERIODS,
CONSIDER THAT A SLOWER CLOCK WILL SAVE POWER AND A 32KHZ WATCH
CRYSTAL IS INEXPENSIVE, BUT REMEMBER THAT THE PROCESSOR EXECUTION
WILL SLOW BY 122 TIMES! IF YOU HAVE A LOT OF PROCESSING TO DO
BETWEEN UPDATES, YOU MAY FIND THE PROCESSOR TOO SLOW!

SOME OTHER TIME OPTIONS:
- 5 BYTES WILL TIME UP TO 25.45 DAYS
- 6 BYTES WILL TIME UP TO 17.83 YEARS
- 7 BYTES WILL TIME 4,566 YEARS!

NO RESET INITIALIZATION IS REQUIRED. THE TIMED PULSE WILL BE
DRIVEN ON THE TCMP PIN WHICH IS AUTOMATICALLY INITIALIZED AS
AN OUTPUT. THE TIMER OUTPUT COMPARE AND THE TIMER OVERFLOW
INTERRUPTS ARE INITIALIZED BY THE START PULSE SUBROUTINE (STRT_PLS).

FLAGS RMB 1 STORE A FLAG
FIRE EQU 7 INDICATES PULSE HAS STARTED
LAST EQU 6 INDICATES LAST INTERRUPT HAS OCCURRED

MAIN PROGRAM GOES HERE. THE LENGTH OF THE DESIRED PULSE IS
DETERMINED AND STORED IN 'PULSE' AT 2 MICROSECONDS PER BIT.
THE PULSE WILL START AFTER 'STRT_PLS' IS CALLED WITH THE
LATENCY AND ACCURACY NOTED BELOW.

RESET ROUTINE

RESET ROUTINE
ORG $100
RST_INT:
CLR TCR
CLR FLAGS RESET ALL FLAGS
LDA #$FF
STA DDR A
LDA #$02
STA PORT A

MAIN PROGRAM

MAIN:
BRSET FIRE, FLAGS, FIRED

LOAD #3 LOAD FOUR BYTES
LDA DO_PLS, X
STA PULSE, X
DECX
BPL LOAD
JSR STRT_PLS

* DURING THE INTERRUPTS, THE 'LAST' FLAG IS CLEAR, JUMP TO MAIN
*
FIRED  BRCLR LAST,FLAGS,MAIN
*
* HERE AFTER THE INTERRUPTS
*
NOP        REPRESENTS OTHER INSTRUCTIONS
BRA  MAIN

*********************************************************************
*                      START PULSE SUBROUTINE                      *
*********************************************************************
*
* CALL THIS ROUTINE WITH THE DESIRED PULSE LENGTH IN 'PULSE'.      *
* THE MOST SIGNIFICANT BYTE IS STORED FIRST. FOR LONG PULSES,      *
* THE 'FRACTIONAL' PART, THAT STORED IN THE TWO LEAST SIGNIFICANT *
* BYTES, ARE TIMED FIRST. THEN THE EXTENSIONS ARE TIMED OUT ONE     *
* AT A TIME UNTIL, ON THE LAST PERIOD THE OUTPUT LEVEL BIT IS     *
* CLEARED AND THE PULSE STOPS AUTOMATICALLY.                      *
*
* NOTE THAT THE VARIABLE PULSE IS MODIFIED BY THE PULSE GENERATION *
* FUNCTION, AND THAT THAT VARIABLE REFLECTS (ROUGHLY) THE AMOUNT   *
* OF PULSE REMAINING. OVERWRITING THE PULSE WIDTH CAN HAVE         *
* UNDESIREABLE RESULTS, BUT SHOULD USUALLY RESULT IN CHANGING THE  *
* TERMINATION TIME.                                               *
*
* PROCEDURE START_PULSE (PULSE_WIDTH: LONG_INT);
*
STRT_PLS:
SEI    DON'T INTERRUPT
*
* IF PULSE_WIDTH > $FFFF THEN INTERRUPT:=ENABLE ELSE INTERRUPT:=DISABLE
BSET   7,PORTA TURN ON INDICATOR LED, NOT TRUE PULSE
*
BSET   OCIE,TCR ENABLE TOC INTERRUPT
LDA    PULSE+1
SUB    #1
STA    PULSE+1
LDA    PULSE
SBC    #0
STA    PULSE
BCC    SP1

*
* HERE IF PULSE WAS LESS THAN $FFFF--FIX THE DAMAGE
*
CLR    PULSE+1
CLR    PULSE
BCLR   OCIE,TCR
*
* IF 0 < PULSE_WIDTH < MINIMUM THEN PULSE_WIDTH := MINIMUM:
*
SP1    TST   PULSE+2
BNE    LONG_PLS
LDA    #PULSE+3
BEQ    LONG_PLS
CMP    MIN_PLS
BHI    LONG_PLS
LDA    MIN_PLS
STA    PULSE+3

LONG_PLS:
* HERE WHEN THE PULSE WIDTH FRACTIONAL PART IS ZERO OR >= MIN_PLS
* FIRST START THE PULSE
* NEXT LEVEL := TRUE;
  BSET    OLVL,TCR

* ONE OF THE TRICKIEST OPERATIONS IS TURNING ON THE PULSE. SINCE
* THE 'HC05 DOES NOT HAVE THE FACILITY TO SWITCH THE TCMP LINE
* DIRECTLY, WE SETUP A TURN ON TO OCCUR IMMEDIATELY. WE HAVE TO
* ADJUST TO THE TIME NEEDED FOR THE SETUP. THIS IS THE VALUE 'DELAY'.
* OUTPUT_COMPARE := TIMER + DELAY
  LDX    ACHR    MUST BE READ FIRST
  LDA    ACLR    TIMER = X:A
  ADD    DELAY
  BCC    MARK_1  MARK TIME
  INCX
  BRA    OC1
MARK_1 NOP    TO BALANCE EXECUTION TIMES
BRA    OC1
OC1  STX    OCHR    INHIBITS TOC
STA    OCLR    ENABLES TOC

* IF DELAY IS CORRECT, PULSE WILL TURN ON IMMEDIATELY
*
*
* TOC := TURN_ON + PULSE_WIDTH MOD $10000
*
  ADD    PULSE+3
  STA    PULSE+3
  TXA
  ADC    PULSE+2
  TAX
  LDA    PULSE+3
  CLR    PULSE+3
  CLR    PULSE+2

* IF INTERRUPT=ENABLED THEN OLVL := 1 ELSE OLVL := 0 ;...AND PULSE
* WILL TERMINATE
*
  BRSET    OCIE,TCR,OC2
  BCLR    OLVL,TCR IF INTERRUPT = DISABLED
  OC2  STX    OCHR
  TST    TSR    WILL CLEAR OCF...
  STA    OCLR    ...WHEN EXECUTED
BSET FIRE,FLAGS  INDICATE PULSE HAS FIRED

*  AT THIS TIME, THE MINIMUM PULSE CAN EXPIRE. IN THAT CASE
*  WHEN WE ENABLE THE INTERRUPT, WE WILL IMMEDIATELY BEGIN
*  SERVICING.
*  CLI
  RTS

**************************************************************************
*  TIMER INTERRUPT ROUTINE  *
**************************************************************************

TCMP_INT:
*
*  WE WILL INTERRUPT WITH A TOC ONLY IF THERE ARE A WHOLE NUMBER OF
*  $10000 PERIODS TO COMPLETE. WE NEED ONLY DECREMENT THE 'INTEGER'
*  PART OF THE PULSE WIDTH AND IF THIS IS THE LAST TIME, WE CLEAR
*  THE INTERRUPTS AND SET THE OUTPUT LEVEL TO '0'. THE TOC REGISTER
*  IS NOT CHANGED.
*  *
*  IF THERE ARE OTHER POSSIBLE TIMER INTERRUPT SOURCES (INPUT CAPTURE
*  AND/OR TIMER OVERFLOW) THEN WE SHOULD ARBITRATE THE SOURCE AT THIS
*  TIME. NOTE THAT THERE WILL ALWAYS BE PLENTY OF TIME TO SERVICE THIS
*  ROUTINE, SO THE PRIORITY COULD BE SET TO THE LOWEST LEVEL.
*  *
*  *
*  ARBITRATION...
  *
*  IF PULSE_WIDTH > $10000 THEN
*  PULSE_WIDTH := PULSE_WIDTH - $10000
*
*     LDA   PORTA
*     EOR   #$03
*     STA   PORTA   TOGGLE 2 PORT LINES (DIAGNOSTICS)
*     *
*     LDA   PULSE+1
*     SUB   #1
*     STA   PULSE+1
*     LDA   PULSE
*     SBC   #0
*     STA   PULSE
*     BCC   NOT_LAST
*  ....ELSE INTERRUPT := DISABLE; OLVL := 0;
*  *
*  HERE IF PULSE WAS ON LAST COUNT, CLEAR INTERRUPT AND OLVL
*  *
*     CLR   PULSE+1
*     CLR   PULSE
*     BCLR  7,PORTA
*     BCLR  OCIE,TCR
*     BCLR  OLVL,TCR
*     BSET  LAST,FLAGS

605
* HERE IF NOT ON LAST PULSE
* CLEAR(OCF);

NOT_LAST:

LDA TSR NECESSARY ACCESS
LDA OCLR ... NEXT INTERRUPT WILL HAPPEN IN $10000
RTI

*********************************************************************
* DUMMY INTERRUPT ROUTINES
*********************************************************************

SPI_INT RTI
SCI_INT RTI
IRQ_INT RTI
SWI_INT RTI

*********************************************************************
* INTERRUPT VECTORS
*********************************************************************

ORG $1FF4
SPI_VEC FDB SPI_INT
SCI_VEC FDB SCI_INT
TIM_VEC FDB TIM_INT
IRQ_VEC FDB IRQ_INT
SWI_VEC FDB SWI_INT
RST_VEC FDB RST_INT
APPENDIX B
LONG PULSE DETECTION

* * *
TIC-TOC ROUTINES FOR 68HC05CX

* * *
WRITTEN 11/18/89 BY MIKE PAUWELS

* * *

PULSE DETECTION

* * *
THIS ROUTINE DETECTS PULSES WITH A MC68HC05CX MICROCONTROLLER USING
THE TIMER INPUT CAPTURE FUNCTION. THE LENGTH OF PULSES DETECTED
CAN RANGE FROM A FEW MICROSECONDS TO MORE THAN TWO HOURS.

* *
THIS SOFTWARE IS INTENDED AS A SUBSYSTEM TO BE INCLUDED IN A LARGER
PROGRAM. ETC.

* *

* CONSTANTS:
OPT no1
* SYSTEM CONSTANTS:
* ADDRESSES:

DDRA EQU $04
PORTA EQU $00
TCR EQU $12
ICIE EQU 7
OCIE EQU 6
TOIE EQU 5
IEDG EQU 1
OLVL EQU 0
TSR EQU $13
ICF EQU 7
OCF EQU 6
TOF EQU 5
ICH R EQU $14
ICLR EQU $15
OCHR EQU $16
OCCLR EQU $17
CHR EQU $18
CLR EQU $19
ACH R EQU $1A
ACL R EQU $1B
OPT 1

* PROGRAM CONSTANTS
* VARIABLES

ORG $BA
AC_OVFL RMB 2
PULSE_W RMB 2
START_T RMB 2

OR CONCATENATE WITH USER MEMORY
MAX TIME = 143.1655765 MINUTES!
HOLDS STOP TIME AND TOTAL PULSE
HOLDS PULSE START TIME
* Assuming a 4 MHz crystal, two bytes can accumulate up to
* 2.33 microseconds (about 2.4 hours) within the accuracy of the
* crystal. Each bit is 2 microseconds. For long time periods,
* consider a slower clock.
* *
* Some other time options:
* 3 bytes will time up to 25.45 days
* 4 bytes will time up to 17.83 years
* 5 bytes will time 4,566 years!
* *
** Flags **
<table>
<thead>
<tr>
<th>Flag</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMB</td>
<td>1</td>
</tr>
<tr>
<td>ARM</td>
<td>7</td>
</tr>
<tr>
<td>GOT</td>
<td>6</td>
</tr>
<tr>
<td>ORG</td>
<td>$100</td>
</tr>
</tbody>
</table>

*******************************************
* RESET INTERRUPT ROUTINE
* ******************************************

RST_INT:
* No reset initialization is required. To measure a pulse incident
* on the input capture pin, arm the procedure by calling ‘GET_PLS’.
* After the pulse is terminated, additional user code (e.g. to set
* a flag) can be added as indicated in the interrupt routine. Note
* that this function requires the interrupt structure to service
* timer overflows and final pulse termination. This is not essential
* and the interrupt structure could be replaced by polling in the
* user’s main loop, as long as the polling period was less than the
* overflow time of the counter/timer.
* *
* BSET 7, DDRA
* BSET 7, PORTA
* CLR   FLAGS
*
* Do other init stuff. The following delay represents other code,
* and gives the LED a momentary flash.
* *
* LDA   $100
* JSR   DELAY       for 1 second
*
* continue
*
*******************************************
* MAIN LOOP
* ******************************************

MAIN:
* BRSET ARM_FLAGS, ARMED
* BSR   GET_PLS
* ARMED NOP
* BRSET GOT_FLAGS, GOT_IT
* NOP
* GOT_IT NOP
* BRA   MAIN

** Flags **
<table>
<thead>
<tr>
<th>Flag</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMB</td>
<td>1</td>
</tr>
<tr>
<td>ARM</td>
<td>7</td>
</tr>
<tr>
<td>GOT</td>
<td>6</td>
</tr>
<tr>
<td>ORG</td>
<td>$100</td>
</tr>
</tbody>
</table>
CALL THIS ROUTINE TO ARM THE PULSE MEASUREMENT. NOTE THAT THE
LENGTH OF PULSE THAT CAN BE MEASURED IS LIMITED BY SIZE OF THE
OVERFLOW ACCUMULATOR. POSITIVE GOING PULSE IS ASSUMED; THE
MODIFICATIONS FOR NEGATIVE GOING PULSE ARE SIMPLY THE INVERSION
OF THE IEDG. SYSTEM IS ARMED 22 MICROCYCLES AFTER THE ROUTINE
IS CALLED.

GET_PLS:
  BSET       IEDG,TCR
  LDA        TSR
  LDA        ICLR
  BSET       ICIE,TCR
  BSET       TOIE,TCR
  START COUNTING OVERFLOWS
  BCLR       7,PORTA
  BSET       ARM,FLAGS
  CLI
  RTS

CALLED FOR A BUSY DELAY. IF NOT INTERRUPTED, WILL RETURN AFTER
A DELAY OF 5 MILLISECONDS TIMES THE CONTENTS OF 'A' ACCUMULATOR.

DELAY:
  LDX        #249
  DLA1
  DECX
  NOP
  NOP
  NOP
  NOP
  NOP
  NOP
  BNE       DLA1
  DECA
  BNE       DELAY
  RTS

HERE ON TIMER INTERRUPT. WE ASSUME THAT TIMER OUTPUT ROUTINES
DO NOT HAVE TO BE ARBITRATED. IF TOC IS NEEDED, THE ARBITRATION
MUST BE CALCULATED. SINCE THE ONLY STICKY PROBLEM OCCURS ON
SIMULTANEOUS OR NEAR-SIMULTANEOUS INTERRUPTS, THE TIMING OF THIS
* ROUTINE IS CAREFULLY CALCULATED.

* TIM_INT:
* THE FOLLOWING INSTRUCTION IS NEEDED IF ANY OTHER TIMER
* INTERRUPTS ARE ENABLED:
* BRCLR ICF,TSR,NO_TIC BR IF NO INPUT CAPTURE
* HERE ON INPUT CAPTURE. IS THIS FIRST EDGE OR LAST EDGE?
* BRCLR IEDG,TCR,LAST_EDG
  BCLR IEDG,TCR PREPARE FOR TRAILING EDGE
* HERE ON THE FIRST (RISING) EDGE
* LDA ICHR
  LDX ICLR <<<< point A
  STA START_T START TIME HIGH BYTE
  STX START_T+1 " " LOW "
* WE NOW HAVE THE CAPTURED START TIME IN MEMORY.
* CLI
  RTI
* HERE ON THE TRAILING EDGE OF THE MEASURED PULSE. THE TIC REGISTER
* HAS THE TWO LEAST SIGNIFICANT BYTES OF THE STOP TIME. SUBTRACT
* THE START TIME; IF NECESSARY BORROW FROM THE AC_OVFL. NO CHECK IS
* MADE FOR OVERFLOW OF THE MAXIMUM PULSE.
* LAST_EDG:
  BSET GOT,FLAGS
  BSET 7,PORTA
  BCLR ICIE,TCR
  LDX ICHR
  LDA ICHR
  STX PULSE_W
  STA PULSE_W+1
* HERE THE PROBLEM IS TOO MANY OVERFLOWS. IF ICHR = $FF AND ACHR = 0
* AND THE OVERFLOW FLAG HAS BEEN CLEARED, WE ACCUMULATED ONE TOO
* MANY OVERFLOW.
* INCA
  TEST FOR = $FF
  BNE CALC_PW
  TST ACHR
  BNE CLEAR_A1
  BRSET TOF,TSR,CLEAR_A1
  LDA AC_OVFL+1
  SUB #1
  STA AC_OVFL+1
  BCC CLEAR_A1
  DEC AC_OVFL

CLEAR_A1:
  LDA ACLR TO CLEAR LATCH
CALC_PW:
  LDA PULSE_W+1

610
SUB       START_T+1
STA       PULSE_W+1
LDA       PULSE_W
SBC       START_T
STA       PULSE_W
LDA       AC_OVFL+1
SBC       #0
STA       AC_OVFL+1
BCC       TIM_EXIT
DEC       AC_OVFL

NO_TIC:

* COULD BE A TOC. OTHER TIC OR TOC OR OVERFLOW STUFF CAN BE DONE HERE
*

* TIM_EXIT:
    RTI

********************************************************************
* DUMMY INTERRUPT ROUTINES
********************************************************************
SPI_INT   RTI
SCI_INT   RTI
IRQ_INT   RTI
SWI_INT   RTI

********************************************************************
* INTERRUPT VECTORS
********************************************************************

ORG       $1FF4
* Interrupt Vectors
SPI_VEC   FDB   SPI_INT
SCI_VEC   FDB   SCI_INT
TIM_VEC   FDB   TIM_INT
IRQ_VEC   FDB   IRQ_INT
SWI_VEC   FDB   SWI_INT
RST_VEC   FDB   RST_INT
Low Skew Clock Drivers and their System Design Considerations

Prepared by Chris Hanke, CMOS Design Engineer
Gary Tharalson, CMOS/TTL Product Planning Manager

ABSTRACT

Several varieties of clock drivers with 1 ns or less skew from output-to-output are available from Motorola. Microprocessor-based systems are now running at 33 MHz and beyond, and system clock distribution at these frequencies mandate the use of low skew clock drivers. Unfortunately, just plugging a high performance clock driver into a system does not guarantee trouble free operation. Only careful board layout and consideration of system noise issues can guarantee reliable clock distribution. This application note addresses these system design issues to help ensure that Motorola’s low skew clock drivers are used effectively in a system environment.

INTRODUCTION

With frequencies regularly reaching 33 MHz and approaching 40–50 MHz in today’s CISC and RISC microprocessor systems, well controlled and precise clock signals are required to maintain a synchronous system. Many microprocessors also require input clock duty cycles very close to 50%. These stringent timing requirements mandate the use of specially designed, low skew clock distribution circuits or ‘clock drivers.’ However, just plugging one of these parts into your board does not ensure a trouble free system. Careful system and board design techniques must be used in conjunction with a low skew clock driver to meet system timing requirements and provide clean clock signals.

Why are Low Skew Clock Drivers Necessary

An MPU system designer wants to utilize as much of a clock cycle as possible without adding unnecessary timing guardbands. Propagation delays of peripheral logic do not scale with frequency. Therefore, as the clock period decreases, the system designer has less time but the same logic delays to accomplish the function. How can he get more time? A viable option is to use a special clock source that addresses these system design issues to help ensure that Motorola’s low skew clock drivers are used effectively in a system environment.

DEFINITIONS

A typical clock driver has a single input which is usually driven by a crystal oscillator. The clock driver can have any number of outputs which have a certain frequency relationship to the clock input. Clock driver skew is typically defined by three different specs. These specs are graphically illustrated in Figure 1.

The first spec, t\text{PS}, measures the difference between the fastest and slowest propagation delays (any transition) between the outputs of a single part. This number must be 1 ns or less for high-end systems.

The second, t\text{PS}, measures the difference between the high-to-low and low-to-high transition for a single output (pin). This spec defines how close to a 50% duty cycle the outputs of the clock driver will be. For example, if this spec is 1 ns (± 0.5 ns), at 33 MHz the output duty cycle is 50% ± 3.5%. A clock driver which only buffers the crystal input, creating a 1:1 input to output frequency relationship, can be a problem if a very tight tolerance to a 50% duty cycle is required. In this situation the output duty cycle is clearly dependent on the input duty cycle, which is not well controlled in most crystal oscillators. The clock driver’s outputs switching at half the input frequency (≈ 2) is a common relationship, which means

17% of a cycle is required for clock distribution or clock ‘uncertainty,’ which is an unacceptable penalty from a system designer’s point of view. At 50 MHz this penalty becomes 25%. A maximum of 10% of the period allotted for clock distribution is an acceptable standard.

If multiple levels of clock distribution (one clock driver’s output feeding the inputs of several other clock drivers) are necessary due to large clock fan-outs, the additional part-to-part skew variations add even more to the clock uncertainty. Standard logic has always been specified with a large (and conservative) delta between the minimum and maximum propagation delays. This delta creates the excessive amount of clock ‘uncertainty’ which the system designer has been forced to design into his system, even though it is not realistic. When system frequencies were below 16 MHz this large clock penalty could be tolerated, but as the above example points out, not anymore. A clock driver’s specs guarantee this min/max delta to be a specific, small value. To reduce the clock overhead to manageable levels, a clock driver with minimal variation (<5%) from a 50% duty cycle and guaranteed low output-to-output and part-to-part skew must be used.
that the outputs switch on only one edge of the oscillator, eliminating the output’s dependence on the duty cycle of the input (crystal oscillator frequency is very stable).

The third spec, $t_{pv}$, measures the maximum propagation delay delta between any given pin on any part. This spec defines the part to part variation between any clock driver (of the same device type) which is ever shipped. This number reflects the process variation inherent in any technology. For CMOS, this spec is usually 3 ns or less. High performance ECL technologies can bring this number down into the 1-2 ns range. Another way to minimize the part-to-part variation is to use a phase-locked loop clock driver, which are just now becoming available.

An important consideration when designing a clock driver into a system is that the skew specs described above are usually specified at a fixed, lumped capacitive load. In a real system environment the clock lines usually have various loads distributed over several inches of PCB trace which can contribute additional delay and sometimes act like transmission lines, so the system designer must use careful board layout techniques to minimize the total system skew. In other words, just plugging a low skew clock driver into a board will not solve all your timing problems.

**DESIGN CONSIDERATIONS**

Figure 2 is a scale replication of a section of an actual 88000 RISC system board layout. The section shown in the figure includes the MC88100 MPU and the MC88200 CMMU devices and the MC88914 CMOS clock driver. The only PCB traces shown are the clock output traces from the MC88914 to the various loads. For this clock driver the output-to-output skew ($t_{OS}$) is guaranteed to be less than 1 ns at any given temperature, supply voltage, and fixed load up to 50 pF.

In calculating the total system skew, the difference in clock PCB trace length and loading must be taken into account. For an unloaded PCB trace, the signal delay per unit length, $t_{pd}$, is dependent only on the dielectric constant, $e_r$, of the board material. The characteristic impedance, $Z_0$, of the line is dependent upon $e_r$ and the geometry of the trace. These relationships are depicted in Figure 3 for a microstrip line. The formulas for $t_{pd}$ and $Z_0$ are slightly different for other types of strip lines, but for simplicity's sake all calculations in this article will assume a microstrip line.

The equations in Figure 3 are valid only for an unloaded trace; loading down a line will increase its delay and lower its impedance. The signal propagation delay ($t_{pd}'$) and characteristic impedance ($Z_0'$) due to a loaded trace are calculated by the following formulas:

$$t_{pd}' = t_{pd} \sqrt{1 + \frac{C_d}{C_o}}$$

$$Z_0' = \frac{Z_0}{\sqrt{1 + \frac{C_d}{C_o}}}$$

$C_d$ is the distributed load capacitance per unit length, which is the total input capacitance of the receiving devices divided by the length of the trace. $C_o$ is the intrinsic capacitance of the trace, which is defined as:

$$C_o = \frac{t_{pd}}{Z_0}$$

Assuming typical microstrip dimensions and characteristics as $w = 0.01$ in., $t = 0.002$ in., $h = 0.012$ in., and $e_r = 4.7$, the equations of Figure 3 yield $Z_0 = 69.4 \Omega$ and $t_{pd} = 0.144$ ns/in. $C_o$ is then calculated as 2.075 pF/in. If it is assumed that an MC88100 or 88200 clock input load is 15 pF, and that two of these loads, in addition to a 7 pF FAST TTL load, are distributed along a 9.6 in. clock trace, $C_d = (2 \times 15 + 7)$ pF/9.6 in. = 3.85 pF/in. The loaded trace propagation delay and characteristic impedance are then calculated as $t_{pd}' = 0.243$ ns/in. and $Z_0' = 41 \Omega$.

Looking at trace c in Figure 2, the two MC88200's are approximately 3 inches apart. Using the calculated value of
The clock signal skew due to the trace is about 0.7 ns. Since these two devices are on the same trace, this is the total clock skew between these devices. Upon careful inspection of all the clock traces, it can be seen that clock signal skew was accounted for and minimized on this board layout. The longest distance between any 88K devices on a single clock trace is about 4.5 inches, which translates to approximately 1.1 ns of skew. The two 88K devices farthest away from the clock driver (traces a and c), are located at almost exactly the same distance along their respective traces, making the clock skew between them the 1 ns guaranteed from output to output of the clock driver. This means that the worst case clock skew between any two devices on this board is approximately 2.1 ns, which at 33 MHz is 7% of the period. Without careful attention to matching the clock traces on the board, this number could easily exceed 3 ns and the 10% cut-off point, even if a low skew clock driver is used.

**CLOCK SIGNAL TERMINATIONS**

Transmission line effects occur when a large mismatch is present between the characteristic impedance of the line and the input or output impedances of the receiving or driving device. The basic guidelines used to determine if a PCB trace needs to be examined for transmission line effects is that if the smaller of the driving device's rise or fall time is less than three times the propagation delay of a switching wave through a trace, the transmission line effects will be present. This relationship can be stated in equation form as:

\[
3 \times t_{pd} \times x \text{ trace length} \leq t_{RISE} \text{ or } t_{FALL}
\]

For the MC88914 CMOS clock driver described in this article, rise and fall times are typically 1.5 ns or less (from 20% to 80% of \(V_{CC}\)). Analyzing the clock trace characteristics presented earlier for transmission line effects, \(3 \times 0.243 \text{ ns/in.} \times \text{trace length} \leq 1 \text{ ns (1 ns is used as 'fastest' rise or fall time). Therefore the trace length must be less than 1.5 inches for the transmission line effects to be masked by the rise and fall times.}

---

**Figure 2. Scale Representation of an Actual 88000 System PCB Layout**

(only sections of the board related to the clock driver outputs are shown).

**Figure 3. Formulas for the Characteristic Impedance and Propagation Delay of a Microstrip Line.** (Ref. 1)
Figure 4 shows the clock signal waveform seen at the receiver end of an unterminated 0.5 inch trace and an unterminated 9 inch trace. These results were obtained using SPICE simulations, which may not be exact, but are adequate to predict trends and for comparison purposes. The 9 inch trace, which is well beyond the 1.5 inch limit where transmission line effects come into play, exhibits unacceptable switching characteristics caused by reflections going back and forth on the trace. Even the 0.5 inch line exhibits substantial overshoot and undershoot. Any unterminated switching characteristics caused by reflections going back and forth on the trace. Even the 0.5 inch line exhibits substantial overshoot and undershoot. Any unterminated line will exhibit some overshoot and undershoot at these edge rates.

Clock lines shorter than 1–1.5 inches are unrealistic on a practical board layout, therefore it is recommended that CMOS clock lines be terminated if the driver has 1–2 ns edge rates. Termination, which is used to more closely match the line to the load or source impedances, has been a fact of life in the ECL world for many years (reference 1 is an excellent source for transmission line theory and practice in ECL systems), but CMOS and TTL devices have only recently reached the speeds and edge rates which require termination. CMOS outputs further complicate the issue by driving from rail to rail (5 V), with slew rates exceeding those of high performance ECL devices.

Since clock lines are only driven from a single location, they lend themselves to termination more easily than bus lines which are commonly driven from multiple locations. Termination of bus lines with multiple drivers is a complicated matter which will not be addressed in this article. The most common types of termination in digital systems are shown in Figure 5. Since no single termination scheme is optimal in all cases, the tradeoffs involving the use of each will be discussed, and recommendations specific to clock drivers will be made. Reference 2 is a comprehensive and practical treatment of transmission line theory and analysis of CMOS signals, and is recommended reading for those who want to gain a better understanding of transmission lines. Figure 6 shows SPICE simulated waveforms of the different termination schemes to be discussed. The driver device in the simulations was the MC88914 output buffer; in all simulations it drove a 9 inch 41 Ω transmission line. The simulations were run using typical model parameters at 25°C and VCC = 5 V.

Series termination, depicted in Figure 5b, is recommended if the load is lumped at the end of the trace and the output impedance of the driving device is less than the loaded characteristic impedance of the trace, or when a minimum number of components is required. The main problem with series termination occurs when the driving device has different output impedance values in the low and high states, which is a problem in TTL and some CMOS devices. A well designed CMOS clock driver should have nearly equal output impedances in the high and low states, avoiding this problem. An additional advantage is that series termination does not create a DC current path, thus the Vol and VOH levels are not degraded. The SPICE generated waveforms of series termination in Figure 6a show that series termination effectively masks the transmission line effects exhibited in Figure 4. If each clock output is driving only one device, series termination would be recommended, but this is not a realistic case in most systems, so series termination is not generally recommended for termination of clock lines.

Parallel termination utilizes a single resistor tied to ground or VCC whose value is equal to the characteristic impedance of the line. Its major disadvantage is the DC current path it creates when the driver is in the high state (if the resistor is tied to ground). This causes excessive power dissipation and VOH level degradation. Since a clock driver output is always switching, the DC current draw argument loses some credibility at higher frequencies because the AC switching current becomes a major component of the overall current. Therefore the main consideration in parallel termination is how much VOH degradation can be tolerated by the receiving devices. Figure 6b demonstrates that this termination technique is effective in minimizing the switching noise, but Thevenin termination has some advantages over parallel termination.

Thevenin termination utilizes one resistor tied to ground and a second tied to VCC. An important consideration when using this type of termination is choosing the resistor values to avoid settling of the voltage between the high and low logic levels of the receiving device. TTL designers commonly use a 220/330 resistor value ratio, but CMOS is a little tricky because the switch point is at VCC/2. With a 1:1 resistor ratio a failure at the driver output would cause the line to settle at

![Figure 4. SPICE Simulation Results of 'Short' and 'Long' Transmission Lines. Simulations were Run with Typical Parameters (α 25°C and VCC = 5 V.](image-url)
2.5 V, causing system debug problems and also potential damage to the receiving devices.

In Thevenin termination, the parallel equivalent value of the two resistors should be equal to the characteristic impedance of the line. A DC path does exist in both the high and low states, but it is not as bad as parallel termination because the resistance in the Thevenin DC path is at least 2 times greater. Figure 6c shows the termination waveforms, which exhibit characteristics similar to parallel termination, but with less $V_{OH}$ degradation. The only real advantage of parallel over Thevenin is less resistors (1/2 as many) and less space taken up on the board by the resistors. If this is not a factor, Thevenin termination is recommended over parallel.

AC termination, shown in Figure 5e, normally utilizes a resistor and capacitor in series to ground. The capacitor blocks DC current flow, but allows the AC signal to flow to ground during switching. The RC time constant of the resistor and capacitor must be greater than twice the loaded line delay. AC termination is recommended because of its low power dissipation and also because of the availability of the resistor and capacitor in single-in-line packages (SIP). A pull-up resistor to $V_{CC}$ is sometimes added to set the DC level at a certain point because of the failure condition described in regards to Thevenin termination. As discussed earlier, the argument of lower DC current is less convincing at high frequencies. The AC terminated waveform walks out slightly toward the end of a high-to-low or low-to-high transition as seen in Figure 6d, making it slightly less desirable than Thevenin termination.

Thevenin and AC termination are the two recommended termination schemes for clock lines, but it depends on what frequency the clock is running at when making a decision between these types of termination. Although hard data is not provided to back this statement up, it is a safe assumption that at frequencies of 25 MHz and below AC is the best choice. If the system frequency could reach 40 MHz and beyond, Thevenin becomes the better choice.

**Additional Considerations when Terminating Clock Lines**

The results presented might imply that terminating the clock lines will completely solve noise problems, but termination can cause secondary problems with some logic devices. Termination acts to reduce the noise seen at the receiver, but that noise actually is seen as additional current and noise at the output of the driving device. If the internal and input logic on the source device is not sufficiently decoupled on chip from the high current outputs, internal threshold problems can occur. This phenomenon is commonly known as 'dynamic threshold.' It is usually evidenced by glitches appearing on the outputs of a fast, high current drive logic device as it switches high or low. This is most severe on 'ACT' devices which have high current and high slew rate CMOS outputs along with TTL inputs which have low noise immunity. This problem can be minimized by decoupling the internal ground and $V_{CC}$ supplies on-chip and in the package. This decoupling is accomplished by having separate 'quiet' ground and $V_{CC}$ pads on chip which supply the input circuitry's ground

![Figure 5. Schematic Representations of Common Termination Techniques](image)
and VCC references. These pads are then tied to extra 'quiet' ground and 'quiet' VCC pins on the package, or to special 'split leads' which resemble a tuning fork and utilize the lead-frame inductance to accomplish the decoupling. When choosing a clock source, make sure that the part has one of these decoupling schemes.

**References**


---

**Figure 6.** SPICE Simulation Results for Various Terminations of a 9 Inch, 41 Ω Transmission Line. Simulations were Run With Typical Model Parameters @ 25°C and VCC = 5.0 V.
Calibration-Free Pressure Sensor System

Prepared by
Michel Burri, Senior System Engineer
Geneva, Switzerland

INTRODUCTION

The MPX2000 Series of pressure transducers are semiconductor devices which give an electrical output signal proportional to the applied pressure. The sensors are a single monolithic silicon diaphragm with strain gage and thin-film resistor networks on the chip. Each chip is laser trimmed for full scale output, offset and temperature compensation.

The purpose of this document is to describe another method of measurement which should facilitate the life of the designer. The MPX2000 Series sensors are available as unported elements and as ported assemblies suitable for pressure, vacuum and differential pressure measurements in the range of 10 kPa through 200 kPa.

The use of the on-chip A/D converter of Motorola's MC68HC05B6 HCMOS MCU makes possible the design of an accurate and reliable pressure measurement system.

SYSTEM ANALYSIS

The measurement system is made up of the pressure sensor, the amplifiers and the MCU. Each element in the chain has their own device-to-device variations and temperature effects which should be analyzed separately. For instance, the 8-bit A/D converter has a quantization error of about ±0.2%. This error should be subtracted from the maximum error specified for the system to find the available error for the rest of elements in the chain. The MPX2000 Series pressure sensors are designed to provide an output sensitivity of 4.0 mV/V excitation voltage with full-scale pressure applied or 20 mV at the excitation voltage of 5.0 Vdc.

An interesting property must be considered to define the configuration of the system, the ratiometric function of both the A/D converter and the pressure sensor device. The ratiometric function of these elements make all voltage variations from the power supply rejected by the system. With this advantage, it is possible to design a chain of amplification where the signal is conditioned in a different way.

The OP-AMP configuration should have a good common-mode rejection ratio to cancel the DC component voltage of the pressure sensor element which is about half the excitation voltage value $V_S$. Also, the OP-AMP configuration is important when the designer's objective is to minimize the calibration procedures which cost time and money and often don't allow the unit-to-unit replacement of devices or modules.

One other aspect is that most of the applications are not affected by inaccuracy in the region 0 kPa thru 40 kPa. Therefore, the goal is to obtain an acceptable tolerance of the system from 40 kPa thru 100 kPa thus minimizing the inherent offset voltage of the pressure sensor.
PRESSURE SENSOR CHARACTERISTIC

Figure 2 shows the differential output voltage of the MPX2100 series at ± 25°C. The dispersion of the output voltage determines the best tolerance that the system may achieve without undertaking a calibration procedure, if any other elements or parameters in the chain do not introduce additional errors.

![Figure 2. Spread of the Output Voltage versus the Applied Pressure at 25°C](image)

The effects of temperature on the full scale output and offset are shown in Figure 3. It is interesting to notice that the offset variation is greater than the full scale output and both have a positive temperature coefficient respectively of +8.0 μV degree and +5.0 μV degree at 5.0 V excitation voltage. That means that the full scale variation may be compensated by modifying the gain somewhere in the chain amplifier by components arranged to produce a negative TC of 250 PPM/°C. The dark area of Figure 3 shows the trend of the compensation which improves the full scale value over the temperature range. In the area of 40 kPa, the compensation acts in the ratio of 40 100 of the value of the offset temperature coefficient.

![Figure 3. Output Voltage versus Temperature. The Dark Area Shows the Trend of the Compensation.](image)

OP-AMP CHARACTERISTICS

For systems with only one power supply, the instrument amplifier configuration shown in Figure 4 is a good solution to monitor the output of a resistive transducer bridge.

The instrument amplifier does provide an excellent CMRR and a symmetrical buffered high input impedance at both non-inverting and inverting terminals. It minimizes the number of the external passive components used to set the gain of the amplifier. Also, it is easy to compensate the temperature variation of the Full Scale Output of the Pressure Sensor by implementing resistors “Rf” having a negative coefficient temperature of -250 PPM/°C.

The differential-mode voltage gain of the instrument amplifier is:

\[
Avd = \frac{V1-V2}{Vs2-Vs4} = \left(1 + \frac{2Rf}{Rg}\right)
\]

![Figure 4. One Power Supply to Excite the Bridge and to Develop a Differential Output Voltage](image)

The major source of errors introduced by the OP-AMP are offset voltages which may be positive or negative and the input bias current which develops a drop voltage V through the feedback resistance Rf. When the OP-AMP input is composed of PNP transistors, the whole characteristic of the transfer function is shifted below the DC component voltage value set by the Pressure Sensor as shown in Figure 5.

The gain of the instrument amplifier is calculated carefully to avoid a saturation of the output voltage and to provide the maximum of differential output voltage available for the A D Converter. The maximum output swing voltage of the amplifiers is also dependent on the bias current which creates a small voltage on the feedback resistance Rf and on the Full Scale output voltage of the pressure sensor.
Figure 5. Instrument Amplifier Transfer Function with Spread of the Device to Device Offset Variation

Figure 5 also shows the transfer function of different instrument amplifiers used in the same application. The same sort of random errors are generated by crossing the inputs of the instrument amplifier. The spread of the differential output voltage \((V1-V2)\) and \((V2x-V1x)\) is due to the unsigned voltage offset and its absolute value. Figures 6 and 7 show the unit-to-unit variations of both the offset and the bias current of the dual OP-AMP MC33078.

Figure 6. Input Offset Voltage versus Temperature

To realize such a system, the designer must provide a calibration procedure which is very time consuming. Some extra potentiometers must be implemented for setting both the offset and the Full Scale Output with a complex temperature compensation network circuit.

The new proposed solution will reduce or eliminate any calibration procedure.

Figure 7. Input Bias Current versus Temperature

MCU CONTRIBUTION

As shown in Figure 5, crossing the instrument amplifier inputs generated their mutual differences which can be computed by the MCU.

Figure 8. Crossing of the Instrument Amplifier Inputs Using a Port of the MCU

Figure 8 shows the analog switches on the front of the instrument amplifier and the total symmetry of the chain. The residual resistance \(RDS(on)\) of the switches does not introduce errors due to the high input impedance of the instrument amplifier.

With the aid of two analog switches, the MCU successively converts the output signals \(V1, V2\).

Four conversions are necessary to compute the final result. First, two conversions of \(V1\) and \(V2\) are executed and stored in the registers \(R1, R2\). Then, the analog switches are commuted in the opposite position and the two last conversions of \(V2x\) and \(V1x\) are executed and stored in the registers \(R2x, R1x\). Then, the MCU computes the following equation:

\[
\text{RESULT} = (R1-R2) + (R2x-R1x)
\]
The result is twice a differential conversion. As demonstrated below, all errors from the instrument amplifier are cancelled. Other averaging techniques may be used to improve the result, but the appropriated algorithm is always determined by the maximum bandwidth of the input signal and the required accuracy of the system.

![Figure 9. Two Channel Inputs and One Output Port are Used by the MCU](image)

**SYSTEM CALCULATION**

<table>
<thead>
<tr>
<th>Sensor out 2</th>
<th>Sensor out 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_{s2} = a(P) + \text{of}_2 )</td>
<td>( V_{s4} = b(P) + \text{of}_4 )</td>
</tr>
</tbody>
</table>

Amplifier out 1  
\[ V_1 = \text{Avd} \left( V_{s2} + \text{OF1} \right) \]

Amplifier out 2  
\[ V_2 = \text{Avd} \left( V_{s4} + \text{OF2} \right) \]

Inverting of the amplifier input  
\[ V_{1x} = \text{Avd} \left( V_{s4} + \text{OF1} \right) \]
\[ V_{2x} = \text{Avd} \left( V_{s2} + \text{OF2} \right) \]

Delta  
\[ \Delta = V_1 - V_2 \]
\[ = \text{Avd} \left( V_{s2} + \text{OF1} \right) - \text{Avd} \left( V_{s4} + \text{OF2} \right) \]

Delta x  
\[ \Delta_x = V_{2x} - V_{1x} \]
\[ = \text{Avd} \left( V_{s2} + \text{OF2} \right) - \text{Avd} \left( V_{s4} + \text{OF1} \right) \]

Adding of the two differential results  
\[ V_{outV} = \Delta + \Delta_x \]
\[ = \text{Avd} \left( V_{s2} + \text{OF2} \right) \]
\[ - \text{Avd} \left( V_{s4} - \text{OF1} \right) \]
\[ = \text{Avd} \left( V_{s2} - V_{s4} \right) \]
\[ + \text{Avd} \left( a(P) + \text{of}_2 \right) - \text{Avd} \left( b(P) + \text{of}_4 \right) \]
\[ = 2 \times \text{Avd} \left( \Delta + \Delta_x \right) \]
\[ = 2 \times \text{Avd} \left( V_{s2} - V_{s4} \right) \]
\[ + \text{Avd} \left( \text{of}_2 - \text{of}_4 \right) \]
\[ + \text{Avd} \left( V_{s2} - V_{s4} \right) \]
\[ + \text{Avd} \left( \text{of}_2 - \text{of}_4 \right) \]
\[ = 2 \times \text{Avd} \left( V_{s2} - V_{s4} \right) \]

There is a full cancellation of the amplifier offset \( \text{OF}_1 \) and \( \text{OF}_2 \). The addition of the two differential results \( V_1 - V_2 \) and \( V_{2x} - V_{1x} \) produce a virtual output voltage \( V_{outV} \) which becomes the applied input voltage to the A/D converter. The result of the conversion is expressed in the number of counts or bits by the ratiometric formula shows below:

\[ \text{count} = V_{outV} \times \frac{255}{\text{VRH-VRV}} \]

255 is the maximum number of counts provided by the A/D converter and VRH-VRV is the reference voltage of the ratiometric A/D converter which is commonly tied to the 5.0 V supply voltage of the MCU.

When the tolerance of the full scale pressure has to be in the range of \( \pm 2.5\% \), the offset of the pressure sensor may be neglected. That means the system does not require any calibration procedure.

The equation of the system transfer is then:
\[ \text{count} = 2 \times \text{Avd} \times \Delta \times 51 \text{V} \]

where:
\( \text{Avd} \) is the differential-mode gain of the instrument amplifier which is calculated using the equation (1). Then with \( R_f = 510 \text{k}\Omega \) and \( R_g = 9.1 \text{k}\Omega \) \text{Avd} = 113.

The maximum counts available in the MCU register at the Full Scale Pressure is:
\[ \text{count (Full Scale)} = 2 \times 113 \times 0.02 \text{V} \times 51 \text{V} = 230 \]

knowing that the MPX2100AP pressure sensor provides 20 mV at 5.0 V excitation voltage and 100 kPa full scale pressure.

The system resolution is 100 kPa 230 that give 0.43 kPa per count.

![Figure 10. Full Scale Output Calibration Using the Reference Voltage VRH-VRV](image)
When the tolerance of the system has to be in the range of ±1%, the designer should provide only one calibration procedure which sets the Full Scale Output (counts) at 25°C 100 kPa or under the local atmospheric pressure conditions.

Due to the high impedance input of the A D converter of the MC68HC05B6 MCU, another configuration may be implemented which uses only one channel input as shown in Figure 11. It is interesting to notice that practically any dual OP-AMP may be used to do the job but a global consideration must be made to optimize the total cost of the system according to the requested specification.

When the Full Scale Pressure has to be set with accuracy, the calibration procedure may be executed in different ways.

For instance, the module may be calibrated directly using Up Down push buttons.

The gain of the chain is set by changing the VRH voltage of the ratiometric A D converter with the R 2R ladder network circuit which is directly driven by the ports of the MCU. (See Figure 12.)

Using a communication bus, the calibration procedure may be executed from a host computer. In both cases, the setting value is stored in the EEROM of the MCU.

The gain may be also set using a potentiometer in place of the resistor Rf. But, this component is expensive, taking into account that it must be stable over the temperature range at long term.

### PRESSURE CONVERSION TABLE

<table>
<thead>
<tr>
<th>Unity</th>
<th>Pa</th>
<th>mbar</th>
<th>Torr</th>
<th>atm</th>
<th>(\text{at = kp/cm}^2)</th>
<th>mWS</th>
<th>psi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 N m(^2) = 1 Pascal</td>
<td>1</td>
<td>0.01</td>
<td>7.5</td>
<td>10</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1 mbar</td>
<td>100</td>
<td>1</td>
<td>0.75</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.014</td>
</tr>
<tr>
<td>1 Torr = 1 mmHg</td>
<td>133.32</td>
<td>1.333</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0.019</td>
</tr>
<tr>
<td>1 atm (1)</td>
<td>101325</td>
<td>1013.2</td>
<td>760</td>
<td>1</td>
<td>1.033</td>
<td>10.33</td>
<td>14.69</td>
</tr>
<tr>
<td>1 at = 1 kp cm(^2) (2)</td>
<td>98066.5</td>
<td>981</td>
<td>735.6</td>
<td>0.97</td>
<td>1</td>
<td>10</td>
<td>14.22</td>
</tr>
<tr>
<td>1 m of water</td>
<td>9806.65</td>
<td>98.1</td>
<td>73.56</td>
<td>0.097</td>
<td>0.1</td>
<td>1</td>
<td>1.422</td>
</tr>
<tr>
<td>1 lb sqin = 1 psi</td>
<td>6894.8</td>
<td>68.95</td>
<td>51.71</td>
<td>0.068</td>
<td>—</td>
<td>—</td>
<td>1</td>
</tr>
</tbody>
</table>

(1) Normal atmosphere  (2) Technical atmosphere
Interfacing Power MOSFETs to Logic Devices

Prepared by Ken Berringer
Motorola Discrete Applications

POWER MOSFET DRIVE CHARACTERISTICS

Power MOSFETs are commonly used in switching applications due to their fast switching speeds and low static losses. When driven with sufficient gate voltage, a power MOSFET will turn on and have a very low on-resistance. If the gate voltage is insufficient to bias the Power MOSFET fully on, or excessive drain currents are applied, the power MOSFET will operate in the saturation (pinch-off) region. In other words, a certain gate voltage will support only a limited amount of drain current.

Most of the current crop of fourth generation power MOSFETs require 10 volts of gate drive to support their maximum continuous drain current. This means that 5 volt logic will not provide enough voltage to drive a standard power MOSFET. A new family of Logic Level power MOSFETs are now available that can support their rated drain current with a gate voltage of 5 volts. With the proper considerations, these power MOSFETs may be easily interfaced to most logic families.

Design of the MOSFET's gate drive is dependent on the MOSFET's input capacitance, which is strongly affected by die size. Therefore, selecting the correct device for the application not only minimizes component cost, but it also optimizes switching performance. Static, or DC, losses are determined by the power MOSFET's on-resistance $R_{DS(on)}$, which is a function of junction temperature ($T_J$), gate voltage ($V_{GS}$), and drain current ($I_D$). $R_{DS(on)}$ is typically specified at $I_D$ equal to half the rated drain current, a $V_{GS}$ of 10 volts, and junction temperatures of 25 and 100°C.

The power MOSFET's static losses can be easily calculated in DC or pulsed applications. First, correct the rated $R_{DS(on)}$ for your drain current and estimated operating temperature with the help of the manufacturers' data sheet curves. Then multiply this value times the RMS load current squared [$P_{static} = I_{rms}^2R_{DS(on)}$]. You should choose a power MOSFET with a current rating ($I_D$) and voltage rating ($V_{DSS}$) well above your worst case load conditions. A good rule of thumb is to select a device with twice your worst case RMS drain current and a voltage rating 25% above your worst case drain voltage.

In high frequency applications switching losses are often more significant than static losses. To minimize switching losses you must decrease the switching times. When a power MOSFET is used in switching applications, the gate cannot be modeled as a simple capacitor due to sizable displacement currents in $C_{rss}$, the drain-to-gate capacitor, brought on by large swings in drain-to-gate voltage. As a result, the total input capacitance, $C_{iss}$, varies greatly over the power MOSFET's operating range. $C_{iss}$ can be piecewise modeled as a linear capacitor in order to find first order approximations of switching times.

A better method of calculating switching times is to use gate charge data from the manufacturers' data sheet. Although a power MOSFET is usually thought of as a voltage controlled device, it can be accurately modeled as a charge controlled device. The charge required for a power MOSFET to handle a given current is relatively constant even though its drain-to-gate capacitance ($C_{rss}$) varies drastically with drain-to-gate voltage. The value of $C_{rss}$ may increase 1000% or more over the operating range.

![Figure 1. Driving a Power MOSFET with a Constant Current Source](image1)

When a power MOSFET is driven by a current source as in Figure 1, its gate voltage will be nearly piecewise linear as shown in Figure 2. The three distinct regions are turn on delay ($t_0$ to $t_1$), rise time ($t_1$ to $t_2$), and excess charge time ($t_2$ to $t_3$).

At the end of the turn on delay ($t_1$) the power MOSFET begins to conduct but the drain current is still very small. During the rise time the power MOSFET actually turns on and the drain voltage drops to almost zero. The resistive switching rise time $t_{rise}$ is actually measured as the time it takes for the drain voltage to drop from 90% to 10% of its highest value. It is called rise time referring to the drain current rise time although the voltage is what's usually measured. This time corresponds to the time that $V_{GS}$ remains in the plateau region of Figure 2.

![Figure 2. Gate-to-Source Voltage versus Time for a Current Source Turning On a Power MOSFET](image2)
During the excess charge time (t2 to t3) $R_{DS(on)}$ continues to decrease. This excess charge must be removed during the turn off delay, so driving the gate to an unnecessarily high voltage will increase the total turn off time.

Unlike bipolar transistors, power MOSFETs are majority carrier devices. Without minority injection, power MOSFETs can be turned off just as easily as they are turned on. For identical gate drive currents, rise time will equal fall time. The turn off waveform for a constant gate current will be a mirror image of Figure 2. Note that the turn off delay does not equal turn on delay, it instead corresponds to the turn on excess charge time.

Since the gate current in Figure 2 is constant and equal to the charge per unit time, the horizontal axis can be labeled time or charge. Gate charge data is usually measured using a 1 mA current source which means it will provide 1 nC (nano-Coulomb) of charge in 1 µs. Manufacturers’ data sheets usually include a gate charge chart of $V_{GS}$ vs $Q_g$ with $Q_g$ labeled in nC as in Figure 3. It is important to note that the value of $V_{GS}$ during rise time, also called the plateau voltage, increases with $I_g$ and therefore so does the turn on delay. Also, the amount of charge needed for rise time will vary with the drain supply voltage. This is usually indicated on the gate charge chart by multiple lines for the excess charge region labeled with the corresponding $V_D$.

To determine the switching times using a current source to drive a Power MOSFET, find the charge required for each region using the gate charge chart, Figure 3, and then use the simple equation:

$$t = \frac{Q_g}{I_g} \quad \text{(1)}$$

First find the charge required during the turn on delay region, $Q_{d(on)}$, by drawing a line down from the first inflection point to the horizontal axis of Figure 3. This is the gate charge for the rated or tested $I_g$. If your actual drain current is different than the rated current you may improve accuracy by linearly scaling $Q_{d(on)}$. Now calculate the turn on delay using Equation 1. Next find the gate charge required for rise time ($Q_{rise}$) from the gate charge chart as the distance between the first inflection point and the intersection of the plateau with the line for your expected $V_D$. A typical value is sometimes listed as $Q_{gd}$. This value may be used to calculate both rise and fall times. Next find the intersection point of your maximum $V_{GS}$ and the line corresponding to your $V_D$. This is the total gate charge $Q_g$(total). To find the charge required for turn off delay $Q_{d(off)}$ (and turn on excess charge), subtract $Q_{d(on)}$ and $Q_{rise}$ from $Q_g$(total). A maximum total gate charge $Q_g$(max) is often specified to facilitate worst case design, however this figure sometimes includes a substantial guard-band.

When driving a power MOSFET with a voltage source with a series resistance (Thévenin source), the calculations are a little more complex. During the rise and fall times $V_{GS}$ is relatively constant since all the gate current is used to charge the gate-to-drain capacitor. By Ohm’s law, $I_g$ is therefore also constant and the gate charge chart can be used with Equation 1 to find rise and fall times. During turn on the voltage across the series resistance is the effective source voltage (usually the supply voltage) minus the gate-to-source plateau voltage, $V_{GS}$. During turn off the voltage across the resistor is the plateau voltage minus the effective sink voltage (usually ground). Rise and fall times will therefore typically be different.

Using this information with Equation 1, we can obtain equations for rise and fall time.

$$t_{rise} = \frac{Q_g}{I_g} = \frac{Q_{gd} R_{eff(ON)}}{V_{SOURCE}-V_{GSP}} \quad \text{(2)}$$

and fall time:

$$t_{fall} = \frac{Q_g}{I_g} = \frac{Q_{gd} R_{eff(OFF)}}{V_{GSP}-V_{SINK}} \quad \text{(3)}$$

$V_{GSP}$ is the Power MOSFET’s gate-to-source plateau voltage, $V_{SOURCE}$ is the gate driver’s effective source voltage, $V_{SINK}$ is the gate driver’s effective sink voltage, and $R_{eff}$ is the gate driver’s effective resistance (output resistance). During turn off $V_{SINK}$ may be near zero volts or even a negative voltage.

\[ \text{Figure 3. Gate Charge Chart for the MTP305SE} \]

During the turn on and turn off delays gate current is not constant and gate charge data cannot be used to determine switching speeds. The series resistance and the gate capacitance form a simple RC network; however, the capacitance varies greatly over the operating range. To find the switching times you must determine the capacitance for each region from a capacitance chart like Figure 4. During the turn on delay $V_{DS}$ is near its maximum value, $V_{GS}$ is near zero, and the input capacitance is low. Find the value of $C_{iss}$ in the capacitance curve for your maximum value of $V_{DS}$ and use this capacitance, Point A in Figure 4, to calculate the turn on delay. You can use Equation 4 to approximate the turn on delay time.

$$I_{d(ON)} = R_{eff(ON)} C_{iss(MIN)} \ln \left[ \frac{V_{SOURCE}}{V_{SOURCE}-V_{GSP}} \right] \quad \text{(4)}$$

During the turn off delay $V_{DS}$ will be low and $C_{iss}$ will have a larger value. Find the value of $C_{iss}$ corresponding to minimum $V_{DS}$ and maximum $V_{GS}$. Point B on the capacitance chart. Then use Equation 5 to approximate turn off delay time.

$$I_{d(OFF)} = R_{eff(OFF)} C_{iss(MAX)} \ln \left[ \frac{V_{G(MAX)}-V_{SINK}}{V_{GSP}-V_{SINK}} \right] \quad \text{(5)}$$
VG(max) is the initial gate voltage prior to turn off (usually the supply voltage). R_{off} is the effective series resistance during turn off, and V_s is the effective sink voltage. If V_s is at ground, then the V_{sink} terms will drop out of Equation 5.

\[ \text{Note that the gate charge chart and capacitance curves are related. The slope of the line in the gate charge chart is in voltage per nano-Coloumb. A Farad of capacitance is equal to a Coloumb per volt.} \]

\[ \text{Farad} = \text{coulomb/volt}. \]

Therefore, the reciprocal of the slope is the input capacitance in nano-Farads (1000 pF). However, you should use both charts. The gate charge chart is most useful when the input capacitance varies and the gate current is constant (rise and fall times). The capacitance curve is most useful when the input capacitance is constant and the gate current varies (delay times).

**DIRECT INTERFACE TO STANDARD POWER MOSFETS**

Standard power MOSFETS can be interfaced directly with standard CMOS devices, such as the MC14000 family. This family uses complementary N and P channel FETs for the output stage. Although standard outputs are rated at ±10 mA and buffer outputs are rated at ±45 mA, saturation currents for short circuit conditions are much higher. While a CMOS gate should not be short circuited for long periods of time, if may be safely operated in the saturation region when switching large capacitive loads. A 14049UB inverter buffer can typically source 30 mA and sink 120 mA using a 12 volt supply. If the output current is not limited, the CMOS gate's output will act like a current source. If the output current is limited to less than the saturation currents, the CMOS gate's output will act like a voltage source with a finite output resistance. The MC14000 series family will operate from 3 to 18 volts. The common 12 or 15 volt V_{DD} supply will drive Power MOSFETS nicely.

The 14049UB can be connected directly to a standard power MOSFET such as the MTP3055E as in Figure 5. The MTP3055E is a rugged 12 amp, 60 volt power MOSFET that is very popular in the industry. The gate drive current is not limited by a series resistor and therefore the gate drive current will be equal to the 14049's output saturation currents of +30/−120 mA. Using the gate charge data with Equation 1, we can predict the following switching times.

\[ \text{t}_{\text{ON}} = 2 \text{nC/30 mA} = 67 \text{nsec} \]
\[ \text{t}_{\text{rise}} = 4 \text{nC/30 mA} = 133 \text{nsec} \]
\[ \text{t}_{\text{OFF}} = 6 \text{nC/120 mA} = 50 \text{nsec} \]
\[ \text{t}_{\text{fall}} = 4 \text{nC/120 mA} = 33 \text{nsec} \]

The switching times were measured using the circuit in Figure 5. The actual scope waveforms are shown in Figure 6, and the measured switching times are shown in Table 1.

<p>| Table 1. Switching Times for Standard CMOS Devices Driving an MTP3055E |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|</p>
<table>
<thead>
<tr>
<th>Driver</th>
<th>V_{CC} (Volts)</th>
<th>R_G (\Omega)</th>
<th>t_{on} (ns)</th>
<th>t_{rise} (ns)</th>
<th>t_{off} (ns)</th>
<th>t_{fall} (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4049UB</td>
<td>12</td>
<td>0</td>
<td>50</td>
<td>150</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>4049UB</td>
<td>12</td>
<td>220</td>
<td>60</td>
<td>300</td>
<td>200</td>
<td>150</td>
</tr>
<tr>
<td>4049UB</td>
<td>12</td>
<td>470</td>
<td>100</td>
<td>400</td>
<td>400</td>
<td>300</td>
</tr>
<tr>
<td>4049UB</td>
<td>15</td>
<td>0</td>
<td>40</td>
<td>100</td>
<td>70</td>
<td>40</td>
</tr>
<tr>
<td>4049UB</td>
<td>15</td>
<td>220</td>
<td>50</td>
<td>200</td>
<td>280</td>
<td>120</td>
</tr>
<tr>
<td>4049UB</td>
<td>15</td>
<td>470</td>
<td>75</td>
<td>330</td>
<td>500</td>
<td>420</td>
</tr>
<tr>
<td>4050B</td>
<td>12</td>
<td>0</td>
<td>50</td>
<td>150</td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td>4050B</td>
<td>12</td>
<td>220</td>
<td>60</td>
<td>300</td>
<td>200</td>
<td>150</td>
</tr>
<tr>
<td>4050B</td>
<td>12</td>
<td>470</td>
<td>100</td>
<td>400</td>
<td>400</td>
<td>300</td>
</tr>
<tr>
<td>4069UB</td>
<td>12</td>
<td>0</td>
<td>100</td>
<td>350</td>
<td>340</td>
<td>250</td>
</tr>
<tr>
<td>4069UB</td>
<td>12</td>
<td>220</td>
<td>115</td>
<td>500</td>
<td>380</td>
<td>370</td>
</tr>
<tr>
<td>4069UB</td>
<td>12</td>
<td>470</td>
<td>150</td>
<td>680</td>
<td>530</td>
<td>580</td>
</tr>
<tr>
<td>4069UB x2</td>
<td>12</td>
<td>0</td>
<td>70</td>
<td>260</td>
<td>170</td>
<td>130</td>
</tr>
</tbody>
</table>

The calculated values were fairly accurate for first order approximations considering that the speeds are high enough that circuit parasitics can affect performance. The saturation currents of the '4049 vary from device to device and with the supply voltage V_{DD} and junction temperature. Driving directly from the logic IC will provide the quickest rise and fall times, but these times will vary greatly.

By adding a resistor between the CMOS buffer's output and the gate of the power MOSFET in Figure 5 we can control switching times by limiting gate drive current. However, increasing the gate resistor also increases the power MOSFET's susceptibility to noise and accidental dv/dt turn on. A rapid change in the power MOSFET's drain voltage will cause a voltage to appear on the gate, which may be sufficient to turn
To find the switching times using a gate resistor, use Equations 2 and 3 to find rise and fall times. Here $R_{\text{eff(on/off)}}$ equals the gate resistor $R_g$, plus the MOSFET's output resistance $R_o$. The approximate output resistance of the 4049 is 200 $\Omega$ for turn on and 50 $\Omega$ for turn off. Let $V_{\text{source}}$ equal $V_{DD}$ and $V_{\text{sink}}$ equal zero. Switching times for several gate resistors are summarized in Table 1.

The UB in the MC14049UB stands for "un-buffered". This means that it consists of a single complementary inverter. The additional gate in Figure 5 is used to ensure the power MOSFET driver is itself driven to $V_{DD}$. The input voltage will greatly affect saturation currents and therefore switching times. The MC14050B is a "buffered" non-inverting buffer and consists of two cascaded inverters. It therefore does not invert the signal, and is less susceptible to soft drive conditions. The diodes on the input in Figure 5 clamp the input voltage to ground and $V_{DD}$. Excessive voltage applied to a CMOS input may damage its internal static protection diodes. Voltage in excess of the supply voltage, $V_{DD}$, applied to the output of a CMOS device may cause it to latch-up and destroy itself. Remember to decouple the logic device, as it is drawing substantial currents.

Open collector TTL gates can also be used to drive standard power MOSFETs. However, most open collector output stages were designed for 5 volt operation. Low power Schottky (LS) gates such as the 74LS05 typically have a collector-emitter breakdown voltage of 10 to 15 volts. This makes them unsuitable for operation using a 12 or 15 volt supply. They can be operated from an 8 to 10 volt supply or with an 8 to 10 volt zener clamp on the output; however, long-term reliability of the logic device will suffer.

The 74LS26 was designed to interface to 15 volt logic and has a tested CE breakdown greater than 15 volts. This Quad NAND gate can be used to drive a power MOSFET with a single pull-up resistor, as in Figure 7. Using a 1.5K pull-up with a 12 volt supply will limit the steady state sink current to 8 mA. This is necessary to guarantee the LS26's rated output low voltage $V_{OL}$ of 0.5 volts. Using a smaller pull-up resistor would increase the $V_{OL}$ of the LS26, and consequently increase the drain-to-source leakage current of the power MOSFET in the off state.

During turn on, current is supplied by the pull-up resistor. During turn off the LS26 must sink both the gate current and the pull-up resistor current. The pull-down transistor of an LS output will typically sink about 30 mA. Turn on times can be calculated using Equations 2 and 4 with $R_{\text{eff(on)}}=R_P$ and $V_{\text{source}}=V_P$, where $R_P$ is the pull-up resistor and $V_P$ is the pull-up's supply voltage. Turn off times can be calculated using Equations 3 and 5 with $R_{\text{eff(0ff)}}=R_P$ and $V_{\text{sink}}=V_P-L_{\text{sink}}R_P$ ($V_{\text{sink}}$ may be negative). The equations for $R_{\text{eff(0ff)}}$ and $V_{\text{sink}}$ are the Thevenin equivalent of an ideal constant current source working against a pull-up resistor. The $V_{\text{sink}}$ equation is only valid when the pull-down transistor may be approximated as a constant source. During the turn off delay and fall times, the pull-down transistor provides a nearly constant sink current, since the pull-down transistor's collector-emitter voltage exceeds its $V_{CE(sat)}$ and the base drive current is relatively constant.

The LS26 with a 1.5K pull-up was used to drive a MTP3055E as in Figure 7. Oscilloscope waveforms are shown in Figure 8, and the switching times are summarized in Table 2.

This configuration provides minimum rise and fall times; however, fall times will vary greatly, since the LS26's sink current will vary with temperature and from device to device. A series gate resistor can be used to slow and control turn off. Switching times can again be calculated using Equations 2 through 5. For large gate resistors you may use the following approximations: $R_{\text{eff(on)}}=R_P+R_G$, $V_{\text{source}}=V_P$, $R_{\text{eff(0ff)}}=R_P$, and $V_{\text{sink}}=0.5$ volts. Switching times for several gate resistors are summarized in Table 2.

<table>
<thead>
<tr>
<th>$V_{CC}$ (Volts)</th>
<th>$R_G$ ($\Omega$)</th>
<th>$R_P$ ($\Omega$)</th>
<th>$t_{\text{rise}}$ (ns)</th>
<th>$t_{\text{fall}}$ (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
<td>1500</td>
<td>200</td>
<td>850</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>1800</td>
<td>200</td>
<td>750</td>
</tr>
<tr>
<td>12</td>
<td>1500</td>
<td>1500</td>
<td>450</td>
<td>2000</td>
</tr>
<tr>
<td>12</td>
<td>3000</td>
<td>3000</td>
<td>930</td>
<td>3900</td>
</tr>
</tbody>
</table>
DIRECT INTERFACE TO LOGIC LEVEL POWER MOSFETs

Logic level Power MOSFETs are designed to be easily interfaced to 5 volt logic devices. They have a larger transconductance and a lower threshold voltage than their conventional counterparts. More importantly, \( R_{DS(on)} \) is specified at \( V_{GS}=5 \) volts. Unfortunately most 5 volt logic families do not have 5 volt high output \( (V_{OH}) \) capability. Fast Schottky (FAST) and Low power Schottky (LS) logic have a minimum rated \( V_{OH} \) of 2.7 volts. This means that a pull-up resistor to 5 volts is required to drive Logic Level Power MOSFETs. High speed CMOS (HC) has a \( V_{OH} \) rating of 4.95 volts, and therefore does not need a pull-up resistor.

Figure 9 shows the output stages of HC and LS logic devices. The HC output stage in Figure 9a is identical to the standard CMOS output stage, except that the complementary MOSFETs have been optimized for 5 volt operation. Most HC devices are buffered by additional complementary stages. The LS output stage in Figure 9b uses a totem pole output. The pull-down transistor is biased on by about 500 \( \mu A \) and has a current gain of about 60. This means it can sink a maximum of 30 mA. The 110 \( \Omega \) resistor limits the pull-up transistor’s sink current to about 30 mA when the output is shorted.

Figure 10 shows how to interface HC, LS, and FAST logic to Logic Level Power MOSFETs. Note the input termination and protection circuitry. This is necessary to drive the logic devices with a pulse generator. It is best to drive the Logic Level Power MOSFET driver with a device from the same logic family. When connecting an HC (or any CMOS) device to an off board connector, the diodes should be used for ESD protection.

Figure 11 shows the switching waveforms for the three logic families driving a Logic Level Power MOSFET using the circuits in Figure 10. The measured switching times are in Table 3.

<table>
<thead>
<tr>
<th>Driver</th>
<th>( R_p ) (( \Omega ))</th>
<th>( t_{d(on)} ) (ns)</th>
<th>( t_{rise} ) (ns)</th>
<th>( t_{d(off)} ) (ns)</th>
<th>( t_{fall} ) (ns)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>74HC04</td>
<td>25</td>
<td>120</td>
<td>85</td>
<td>75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>74LS04</td>
<td>560</td>
<td>45</td>
<td>450</td>
<td>120</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>74F04</td>
<td>220</td>
<td>15</td>
<td>170</td>
<td>18</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>74HC04</td>
<td>10</td>
<td>65</td>
<td>30</td>
<td>30</td>
<td></td>
<td>2 gates</td>
</tr>
<tr>
<td>74HC04</td>
<td>10</td>
<td>125</td>
<td>35</td>
<td>45</td>
<td></td>
<td>12A 50°C</td>
</tr>
</tbody>
</table>

Table 3. Switching Times for Logic Devices Driving a Logic Level MTP3055EL

\( I_D \) = 6 Amps unless noted. One gate used unless noted.
current and junction temperature. If low voltage operation is a real possibility you should choose the Logic Level Power MOSFET and heatsink to handle this worst case condition. Examine the curves for "On-region Characteristics", "RDS(on) versus ID", and "RDS(on) versus Temperature" in the manufacturers' data sheet. You may need to use a device with a current rating much larger than your expected load current to attain the desired RDS(on) under low supply conditions. Manufacturers are now developing 4 volt logic level power MOSFETs with RDS(on) rated at 4 volts. These devices may be easily interfaced to HC logic devices and operated down to 4 volts. However, the lower threshold voltage makes them more susceptible to noise and increases leakage currents.

The 74LS04 in Figure 10 must have a pull-up resistor to 5 volts. A minimum pull-up resistor of 560 Ω will guarantee the logic device's output low voltage, VOL, of 0.5 volts. During turn on, gate drive current is supplied by the pull-up resistor and the 'LS04's internal pull-up transistor. During turn off the 'LS04 must sink both the gate drive current and the pull-up resistor current. A larger R P will increase turn on time and decrease turn off time. A smaller R P would increase the VOL of the 'LS04, increasing the power MOSFET's leakage current. The lower threshold voltage of logic level power MOSFETs makes the VOL rating critical. The threshold voltage of a power MOSFET decreases as temperature increases. Therefore, the VOL of the logic device must be less than the logic level power MOSFETs threshold voltage VGS(th) at its maximum expected junction temperature. For this reason 4 volt logic level power MOSFETs may be incompatible with TTL logic devices.

Switching times can again be estimated by using the Thevenin equivalents of the drive circuit with Equations 2 through 5. During turn on delay, current is supplied by the Darlington pull-up transistor of the 74LS04, and the external pull-up resistor. The Darlington is in saturation with a VCE(sat) of about 1.5 Volts. The 74LS04's output current is then limited by the internal 110 Ω resistor. To calculate turn on delay, you may use Equation 4 with Vsource = VCC[1.5RP/(RP + 110Ω)] and R eff(on) = RP[110Ω]. During rise time nearly all the current is supplied by the pull-up resistor, since VGGP is usually above the VOH of the 'LS04. You may therefore use Equation 2 with Vsource = VCC and R eff(on) = RP to estimate rise time.

During turn off the pull-down transistor must sink both the gate current and the pull-up resistor current, just like the open collector 74LS26 in Figure 7. To calculate turn off times, use Vsink = VCC - IsinkRP and R eff(off) = RP with Equations 3 and 5. The pull-down transistor's maximum sink current, Isink, is typically about 30 mA.

The 74LS family's specified supply voltage (VCC) range is from 4.75 to 5.25 volts. The rise time will vary greatly with supply voltage while the fall time only varies by about 5%. The rise time will vary from about +80% to −40% for VCC equals 4.75 and 5.25 volts respectively. This is due to supply voltage affecting both the pull-up resistor current and the pull-up transistor current. Since the operating supply range of LS is less than that of HC logic, RDS(on) will not vary as much, but must be considered.

The FAST logic family can source and sink much more current than the LS family. The 74F04 can source about 50 mA and sink about 200 mA. A minimum pull-up resistor of 220 Ω will guarantee the logic device's output low voltage VOL of
0.5 volts. A larger $R_p$ will increase turn on time and decrease turn off time. The switching times can be calculated as in the LS family. The 74F04 uses an internal 35 $\Omega$ resistor to limit the pull-up Darlington's output current, instead of the 110 $\Omega$ resistor. The same supply voltage considerations for LS family also apply to the FAST family.

A series gate resistor may be used with any of the circuits in Figure 10 to slow and control switching times. The switching times for large gate resistors (greater than 200 $\Omega$ for HC, 5K$\Omega$ for LS, and 2K$\Omega$ for FAST) can be estimated using $R_{on/off} = R_g$ with the Equations 2 through 5: When switching loads even slightly inductive, the inductive kick-back during turn off may cause the drain voltage to rise above the load supply. Slowing down the turn off with a gate resistor will reduce this voltage. If this voltage is large enough and sufficient energy is present it may destroy the Power MOSFET. A new family of rugged Power MOSFETs can handle considerable energy under these conditions. You may also want to choose a large $R_p$ value in order to reduce Electromagnetic Interference (EMI). When driving a lamp, you may want to use a very large resistor to limit in-rush current. Long-term reliability of the logic device will also be improved by using a gate resistor and/or a larger pull-up resistor. The gate resistor dissipates most of the drive power losses, instead of the logic device, reducing stress on the logic output devices. A larger pull-up resistor limits the steady state current in the pull-down transistors, thereby decreasing their power dissipation. However, using a large gate resistor will also increase the power MOSFET's susceptibility to noise and dv/dt turn on.

Logic gates on the same chip may be paralleled to increase switching speeds. The output current capability will increase in proportion to the number of gates used. If no gate resistor is used, the switching times will decrease in proportion to the number of gates used. If a gate resistor is used it may be safely decreased, in proportion to the number of gates, to decrease switching times. Paralleling logic gates will not change the total logic package power dissipation, since the output current increases and switching times decrease. When many gates are used, switching times may decrease to the point where they are limited by the stray inductance in the load and in the lay-out. Logic gates on different chips or from different families should not be paralleled because the different propagation delays may cause excessive shoot-through currents which might damage the logic devices.

Spare gates left over from a digital circuit may be used to drive a Logic Level power MOSFET. However, the large currents being used by the driver may cause large amounts of noise on the supply rail. This noise may cause data errors in the other gates on the same IC. Limiting the current with a large gate resistor and carefully decoupling the logic device will reduce the power supply noise. Also the driving logic device must be grounded at the same point as the source of the power MOSFET to avoid ground shift problems caused by the large drain currents. If separate logic and analog grounds are used they should be connected only at the source of the power MOSFET.

Pay close attention to the power supply scheme. The gate of a power MOSFET should never be left floating with voltage
Figure 11. Logic Devices Driving an MTP3055EL

(a) 74HC04

(b) 74LS04 \( R_P = 560 \ \Omega \)

(c) 74F04 \( R_P = 220 \ \Omega \)
applied to the drain. When this happens the power MOSFET may turn on and destroy itself if the current is not limited. If separate supplies are used for the load and the logic IC, the logic supply should be powered up first and powered down last. If this is not possible, consider what happens to the logic device output when power is removed. The pull-up resistors in the LS and FAST circuits of Figure 10 will pull the power MOSFET’s gate down to VCC when it is low, turning the power MOSFET off. The HC inverter’s output, however, will be in a high impedance state when the logic supply voltage is low, allowing the power MOSFET’s gate to float. A large resistor to the logic supply voltage or ground, or using a small signal diode to clamp the output to below the logic supply voltage, will solve this problem. Low logic supply voltage may also cause power MOSFET failure due to insufficient gate drive. When a power MOSFET fails the drain voltage will usually appear at its gate, which may take out the entire logic circuit. A gate resistor will also limit the current under this power MOSFET failure condition.

INTERFACING TO A MICROPROCESSOR

Microprocessors can be easily interfaced to a Power MOSFET. Any of the circuits in Figure 10 can be used as a buffer between a microprocessor port and a Logic Level power MOSFET. If you want to use a standard power MOSFET, you will have to use the ‘LS26 circuit in Figure 7 or a level shifter. The MC14504B hex level shifter can be used to interface HC, LS, or FAST to standard CMOS. This level shifter can be used to drive the Power MOSFET directly or with a buffer like the MC14049UB in Figure 5 to decrease switching times. The MC14504B has selectable TTL/CMOS level inputs and standard CMOS outputs. It can source and sink a maximum of about 20 mA using a 12 volt supply.

Be very careful when using bus drivers and latches which have tri-state outputs, like the 74LS240-74HC240 and 74LS373-74HC373, to drive a power MOSFET. The LS tri-state devices require a pull-up resistor to drive the power MOSFET to 5 volts, and will therefore leave the power MOSFET on when the outputs are disabled. The HC devices with tri-state outputs will let the gate float when the outputs are disabled, possibly damaging the power MOSFET. Tri-state devices can be used provided the output enable pin is tied true, low for negative logic enable inputs. HC tri-state devices do not require a pull-up resistor to drive a logic level power MOSFET, and may therefore be used with a pull-down resistor to ground. Note that tri-state outputs should never be pulled above the supply rail or below ground.

When simplicity is important, a single chip microcomputer like the 68HC11 can be used to drive a power MOSFET directly. This microcomputer may be used to perform functions like Pulse Width Modulation, complex motor speed control, and controlling multiple power MOSFETs for bridge applications. When the microcomputer is used in the single chip mode, any one of the 8 pins of parallel output port B can be used to drive a Logic Level power MOSFET. A large gate series resistor should be used to minimize power dissipation and noise on the chip. This means that switching times will be fairly slow. This arrangement also exposes the microprocessor to possible harm from power MOSFET failure. Although all the outputs of port B will be reset to zero on a Power-On Reset (POR), a pull down to ground should be used to ensure the power MOSFET will be off during power down. In some applications it may be necessary to initialize the power MOSFET gate drive via software before power is supplied to the power MOSFET.

Port B may also be used in a strobed mode by using the STRB signal from the control port D. The STRB signal will go high after the data on port B is valid and may be used to latch or enable a logic device driving a power MOSFET. This mode may be useful when exact synchronization is desired between the microprocessor controlled devices.

When used in the extended memory mode, ports B and C are used for address and data busses. The 68HC24 port replacement unit will replace port B in a software transparent fashion. Thus, a system can be developed using the 68HC11 with a 68HC24 and external memory, while the final product will use only the 68HC11.

CONCLUSION

We have seen that standard Power MOSFETs can be interfaced directly to standard CMOS logic with very good performance, about 50 ns rise and fall times for the MC14049UB driving a 12 Amp power MOSFET. Standard Power MOSFETs may also be interfaced to 5 volt logic using a special interface device such as the 74LS26 open collector NAND gate or the MC14504B hex level shifter. The 74LS26 driving a 12 Amp standard Power MOSFET gives turn on times of about 1 μs and fast turn off times of less than 200 ns. Switching times may be easily estimated using four simple equations and a series resistor may be selected to give the desired rise and fall times.

Logic Level Power MOSFETs can be driven directly with HC logic, and by LS logic with the addition of a pull-up resistor. Switching speeds using an HC device are very fast, less than 150 ns per gate when driving a 12 Amp power MOSFET. Using an LS device, turn on speed is good, about 0.5 μs, and turn off speed is excellent, less than 150 ns. Again, switching speeds may be easily estimated and a series resistor may be selected to give the desired performance.

Logic power supply variations are the most important aspect affecting Logic Level Power MOSFET performance. Power supply sequencing and under-voltage protection is necessary to ensure system integrity. Circuit lay-out and power supply decoupling are also important at high speeds. Finally a Logic Level Power MOSFET may be interfaced directly to a dedicated microprocessor output port when microprocessor control is desired.

Bibliography

Motorola Power MOSFET Transistor Data, DL135 Rev 2, 1988, Ch 1-4, 6, and pp. 3.711-716.
Motorola FAST and TTL Data, DL121 Rev 3, 1988, Ch 2, pp. 4.6-7 and 5.6.
Motorola High-Speed Logic Data, DL129 Rev 3, 1988, Ch 4 and pp. 5.11-14.
Basic Servo Loop Motor Control Using the MC68HC05B6 MCU

By Jim Gray

This application note describes a basic circuit and software implementing proportional derivative (PD) closed-loop speed control for a brush motor using four integrated circuits (ICs), two opto discretes, and less than 200 bytes of code.

Feedback control systems using digital algorithms implemented on microcontroller units (MCUs) are becoming increasingly commonplace. The use of an MCU in this type of control application is justified when system flexibility is needed, such as varying drive motors or storing wear parameters in electrically erasable programmable read-only memory (EEPROM). Typically, the system would be modeled mathematically in the discrete time domain due to the use of sampled rather than continuous data. The linear difference equations describing the transfer function of the system are solved using z-transforms, allowing, in the case of proportional-integral-derivative (PID) control, the determination of constants for proper system performance and stability. However, this level of analysis is not necessary to illustrate how straightforward the implementation is using the MC68HC05B6 and the MPM3004 TMOS™ H-bridge. The generalized flow of a PD loop is shown in Figure 1. The transfer function of \( G_c(s) \) consists of the PD control, and \( G_p(s) \) represents the power amplifier, motor, and load. Here \( s \) is a complex variable having both real and imaginary parts. The proportional term \( K_p \) can be accomplished with shifting operations, at least to the resolution of powers of 2. The derivative term, \( K_D s \), of \( f(t) \) is approximately

\[
\frac{df(t)}{dt} \bigg|_{t=kT} \equiv \frac{1}{T} [f(kT) - f((k-1)T)]
\]

where \( f(kT) \) is the current value of the controlled parameter, and \( f((k-1)T) \) is the value of the same parameter at the previous sampling time. In this example, \( K_D s \) is realized as the rate of change of the difference between the measured and the desired period of motor-shaft rotation.

![Figure 1. PD Loop Flow](image-url)
The MC68HC05B6 is an M68HC05 MCU Family member with two channels of programmable pulse-length modulation on-chip. When used with an H-bridge device such as the MPM3004, these channels can control bidirectional currents of up to 10-A continuous (25-A peak) at 60 V (see Figure 2). Two I/O pins and both pulse-length modulation (PLM) channels are used to control the MPM3004. Proper gate drive and level conversion is provided by the MC34151 dual inverting gate drivers. Input to the control loop consists of the MLED71 infrared emitter and MRD750 photo Schmitt trigger detector coupled through a slotted disc on the motor shaft. The TCAP2 pin and associated input capture registers are used to convert the optical index marks into a time measurement. Great care must be taken to ensure an adequate current source for the MPM3004 and to isolate the supply for the MC34151s. Separate circuit runs and 0.1-µF bypass capacitors on the MC34151 ICs were used in this case.

The justification for adding a derivative term to a proportional controller can be easily understood by examining the reasons for the overshoot and ringing typical of an underdamped proportional-only controller. When proportional control applies additional power to correct an underspeed condition, it does so continuously until the error term is zero, resulting in a power setting that ensures an overspeed condition. The converse occurs when reducing motor speed. The rate of change of the error signal as excessive power is being applied to correct underspeed will be a relatively large negative value (the error term is being rapidly reduced). Thus, the derivative of the error term is of the correct sign to compensate the proportional gain term. One effect of this compensation is to retard the loop’s response time, but the proportional gain can be increased to offset this.

The listing (see Figure 3) shows the assembly source code for speed measurement and the PD control of PLMA, which drives the power H-bridge in one direction. The opposite direction of rotation is obtained by complementing bits 0 and 1 of port A and driving the opposite lower leg of the H-bridge with PLMB. Eight-bit arithmetic was used exclusively in this example for space and clarity. Although this approach is functional, 16-bit routines for multiply and divide, given in Reference 2, are better for finer control. Routines to set initial values, control direction of rotation, and check for motor stall are also necessary, although they are not shown in this application note.

Figure 4 shows the response of the system to various changes in load. The data was captured in an emulator trace buffer (Motorola CDS8 Jewelbox) and plotted using a data base program. Beginning from a no-load condition at 4 s, loading (an uncalibrated friction brake) was ramped to cause approximately a 50-percent duty cycle. Starting at 10 s, the load was then increased again until the system was at the limit of compliance — i.e., at full power and still maintaining the desired speed. Next, at 14 s, approximately half the load was rapidly (0.1 s) removed. The gain of the proportional term was 2, and the derivative constant was 1. In systems where a low-pass filter would be beneficial or the steady state error is potentially large, an integral term could be added for full PID control.

REFERENCES

Figure 2. Block Diagram of Servo Loop Motor Control
This program performs a closed loop servo speed control using PLMA for output. Speed is measured optically with a slotted disk. The optically detected index mark controls TCAP2 which allows calculation of the period of revolution for the loop input.

**MC68HC05B6 SERVO LOOP MOTOR CONTROL EXAMPLE**

```
org $0

cycles off

org $0A

org $12

TCR RMB 1

TSR RMB 1

CAHR1 RMB 1

CALR1 RMB 1

COHR1 RMB 1

COLR1 RMB 1

CNTHR RMB 1

CNTLR RMB 1

ACNTHR RMB 1

ACNTLR RMB 1

CAHR2 RMB 1

CALR2 RMB 1

PERIOD RMB 1

PLMTMP RMB 1

DESPRD RMB 1

OELTAN RMB 1

OELTAO RMB 1

DELTADC RMB 1

BEGIN LDA #54

SELECT SLOW PLM REPETION RATE

STA MISC

SPEED

LDA PLMTMP

LOAD PLM VALUE

STA PLMA

CLEAR FLAG AND ANY PENDING INT.

LDA CAHR2

LDA CALR2

SET INPUT CAPTURE INTERRUPT ENABLE

7.TCR

CLEAR I BIT ALLOWING TIMER INTERRUPTS

LDA TSR

BSET 7.TCR

WAIT FOR OPTO INDEX TCIC INTERRUPT

LDA TSR

CLR TSR BIT 4 TO ENSURE

LDA CAHR2

SYNCHRONIZATION TO INDEX
```

Figure 3. MC68HC05B6 Servo Loop Motor Control Example
TEST FLAG FOR INDEX1

STORE COUNT

DELAY TO AVOID RETRIGGER ON SAME INDEX

CLEAR FLAG AND WAIT

FOR INDEX2

STORE SECOND COUNT

CALCULATE PERIOD

STORE.

GET PREVIOUS ERROR AND STORE IT.

LOAD DESIRED PERIOD, SUBTRACT ACTUAL

TO FORM DELTAN.

MULTIPLY ERROR BY 2.

OR FALL THRU TO DECREMENTING HERE.

FORM RATE OF CHANGE OF ERROR AND STORE.

GET CURRENT ERROR AND APPLY \(\frac{DE}{DT}\) CORRECTION THEN STORE.

GET CURRENT PLM AND APPLY CORRECTION.

BRANCH TO DECREMENT IF RESULT POSITIVE

OTHERWISE IN LOW SATURATION SO KEEP PLM TEMP MINIMUM.

SEE IF PLM TEMP MINIMUM DECIMENT PLMA

UPDATE PLMA TEMPORARY LOCATION

MULTIPLY ERROR BY 2

IN SATURATION OR CORRECTION EQUALS 0 RETURN TO WAIT

RETURN TO WAIT

set vectors

R

SCI

TOV

TOC

IRQ

SWI
Symbol Table:

<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Value</th>
<th>Def. #</th>
<th>Line Number</th>
<th>Cross Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACNTHR</td>
<td>001A</td>
<td>*00036</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACNTRL</td>
<td>001B</td>
<td>*00037</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADJDN</td>
<td>0F6A</td>
<td>00110</td>
<td>00105</td>
<td></td>
</tr>
<tr>
<td>ADJUP</td>
<td>0F89</td>
<td>*00126</td>
<td>00124</td>
<td></td>
</tr>
<tr>
<td>BCNTH</td>
<td>0050</td>
<td>*00043</td>
<td>00072</td>
<td>00088</td>
</tr>
<tr>
<td>BCNTL</td>
<td>0051</td>
<td>*00044</td>
<td>00074</td>
<td></td>
</tr>
<tr>
<td>BEGIN</td>
<td>0F00</td>
<td>*00056</td>
<td>00132</td>
<td>00133 00134 00135 00137 00138 00139</td>
</tr>
<tr>
<td>CAHR1</td>
<td>0014</td>
<td>*00030</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAHR2</td>
<td>001C</td>
<td>*00038</td>
<td>00061</td>
<td>00067 00071 00079 00083</td>
</tr>
<tr>
<td>CALR1</td>
<td>0015</td>
<td>*00031</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALR2</td>
<td>001D</td>
<td>*00039</td>
<td>00062</td>
<td>00068 00073 00080 00085</td>
</tr>
<tr>
<td>CNTHR</td>
<td>0018</td>
<td>*00034</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNTLR</td>
<td>0019</td>
<td>*00035</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COHR1</td>
<td>0016</td>
<td>*00032</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLR1</td>
<td>0017</td>
<td>*00033</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC1</td>
<td>0F27</td>
<td>*00076</td>
<td>00077</td>
<td></td>
</tr>
<tr>
<td>DECSPD</td>
<td>0F70</td>
<td>*00113</td>
<td>00111</td>
<td></td>
</tr>
<tr>
<td>DELTADC</td>
<td>0059</td>
<td>*00052</td>
<td>00099</td>
<td>00101 00102 00121 00123</td>
</tr>
<tr>
<td>DELTAN</td>
<td>0057</td>
<td>*00050</td>
<td>00090</td>
<td>00096 00098 00100 00104 00117 00119 00120</td>
</tr>
<tr>
<td>DELTAO</td>
<td>0058</td>
<td>*00051</td>
<td>00091</td>
<td>00097 00110</td>
</tr>
<tr>
<td>DESPRD</td>
<td>0056</td>
<td>*00049</td>
<td>00092</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td>0F8D</td>
<td>*00128</td>
<td>00109</td>
<td>00115 00125</td>
</tr>
<tr>
<td>ECNTH</td>
<td>0052</td>
<td>*00045</td>
<td>00084</td>
<td>00087</td>
</tr>
<tr>
<td>ECNTL</td>
<td>0053</td>
<td>*00046</td>
<td>00086</td>
<td></td>
</tr>
<tr>
<td>INCSPD</td>
<td>0F76</td>
<td>*00116</td>
<td>00094</td>
<td></td>
</tr>
<tr>
<td>INDEX1</td>
<td>0F16</td>
<td>*00071</td>
<td>00069</td>
<td></td>
</tr>
<tr>
<td>INDEX2</td>
<td>0F35</td>
<td>*00083</td>
<td>00081</td>
<td></td>
</tr>
<tr>
<td>KEYS</td>
<td>0F08</td>
<td>*00060</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MISC</td>
<td>000C</td>
<td>*00024</td>
<td>00057</td>
<td></td>
</tr>
<tr>
<td>PADDR</td>
<td>0004</td>
<td>*00016</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PADR</td>
<td>0000</td>
<td>*00012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBADDR</td>
<td>0005</td>
<td>*00017</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBDR</td>
<td>0001</td>
<td>*00013</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCADDR</td>
<td>0006</td>
<td>*00018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCDR</td>
<td>0002</td>
<td>*00014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PDIDR</td>
<td>0003</td>
<td>*00015</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERIOD</td>
<td>0054</td>
<td>*00047</td>
<td>00089 00093</td>
<td></td>
</tr>
<tr>
<td>PLMA</td>
<td>000A</td>
<td>*00022</td>
<td>00059</td>
<td>00107 00113 00126</td>
</tr>
<tr>
<td>PLMB</td>
<td>000B</td>
<td>*00023</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLMMIN</td>
<td>0F62</td>
<td>*00106</td>
<td>00112</td>
<td></td>
</tr>
<tr>
<td>PLMTMP</td>
<td>0055</td>
<td>*00048</td>
<td>00058 00103 00108 00114 00122 00127</td>
<td></td>
</tr>
<tr>
<td>RPM</td>
<td>0F13</td>
<td>*00066</td>
<td>00136</td>
<td></td>
</tr>
<tr>
<td>TCR</td>
<td>0012</td>
<td>*00028</td>
<td>00063</td>
<td></td>
</tr>
<tr>
<td>TFLAG1</td>
<td>0F19</td>
<td>*00069</td>
<td>00070</td>
<td></td>
</tr>
<tr>
<td>TFLAG2</td>
<td>0F30</td>
<td>*00081</td>
<td>00082</td>
<td></td>
</tr>
<tr>
<td>TSR</td>
<td>0013</td>
<td>*00029</td>
<td>00060</td>
<td>00066 00069 00078 00081</td>
</tr>
<tr>
<td>WAIT</td>
<td>0F11</td>
<td>*00065</td>
<td>00065</td>
<td></td>
</tr>
</tbody>
</table>

Errors: None
Labels: 47

Last Program Address: $1FFF
Last Storage Address: $FFFF
Program Bytes: $009E 158
Storage Bytes: $0020 32
Figure 4. Step Response of PLM Motor Control
A Software Method for Decoding the Output from the MC14497/MC3373 Combination

Prepared by: Steve Reinhardt

The electronics industry has used infrared media as a simple, easy, and effective method of wireless communications over short distances. It is not without its problems since simple on/off modulation is affected by the many infrared sources in our environment today. To provide immunity from the noise created by lamps, lighters, electronics, and even humans, the IR carrier is modulated at a rate that would not occur in nature. The industry has settled on around 40 kHz as the modulation frequency.

The data that is transmitted usually takes the form of AM (or CW – continuous wave); that is the carrier is turned on and off for variable periods of time. Some have used a FM scheme, where the modulation frequency is changed to represent 1 or 0. The output of detectors is generally the same: that is a logic 0 represents a presence of carrier in AM, or one of the frequencies in FM. A logic 1 then represents no carrier in AM, or the second frequency in FM.

The encoding of the data varies widely, from schemes that encode the data as variable pulse widths, constant length coding schemes, or simple ASCII, to the biphase scheme used in the MC14497. Any of these schemes can be decoded by the use of a microcomputer that has a timer, such as the MC68HC05 family or the MC68HC11 family of parts.

THE MC14497

The MC14497 is a complete building block for IR data transmission, lacking only a high current driver to power the IR LED (or LEDs, depending on the range required) such as the MLED81. The chip limits the duty cycle of the LED to about 10%. The use of an inexpensive ceramic resonator generates the 31.25 kHz carrier. A simple SPST matrix keyboard completes the transmitter.

THE MC3373

The MC3373 is a companion chip to the MC14497. It provides all of the front-end signal processing to interface an IR photo detector, such as the MRD821, to a TTL level. It includes the gain stages, with automatic background level control (AGC), a simple frequency discriminator to eliminate interference from other sources, and a wave shaper that generates a TTL or CMOS output level. The MC3373 does not decode the data, it merely reconstructs it in logic level form (the way it was transmitted) to facilitate decoding. It too requires few outside components, such as a tuned circuit and a few capacitors for wave shaping. Care must be taken in circuit layout, as this device operates at very high gain to accommodate the low level input signal from the photodetector. Since there are frequency sensitive components in the circuit, it is best to minimize lead lengths, work on a ground plane, and perhaps even put a shield around the components that make up the receiver.1

THE ENCODED BIT STREAM

To understand how to decode the data from the MC14497, it is important to understand what is transmitted. Each word transmitted consists of an AGC burst, a start bit, and 6 data bits. The 6 data bits represent 64 individual channels (0–63). However, channel 63 (111111) is never sent.

The carrier frequency is determined by dividing the oscillator frequency by 16. For a 500 kHz resonator, the carrier is 31.25 kHz. The baud rate (signalling rate) is equal to the carrier divided by 32, or the oscillator divided by 512. Again, for a 500 kHz resonator, the baud rate is approximately 976 bps. Each command word takes approximately 8 ms to send.

Refer to Figure 1. Data is the representation of the channel code (001010, channel 10). Carrier represents the output of the MC14497. Recovered is the signal that is output from the MC3373. Notice that the data is inverted, or normally high. When a key is pressed, the chip sends a signal to setup the AGC in the receiver. In the AM mode, which is most common, each transmitted word is preceded by a 512 μs burst (oscillator divided by 256). One bit time later, the start bit is sent. The biphase modulation scheme then uses the position of a carrier within the bit time to represent the data value. Refer to Figure 2. The presence of carrier immediately after the bit time boundary represents a 1. The lack of the carrier represents a 0. The phase then changes so that there is a constant modulation, with clock edges on each bit time boundary. This feature is important in some communications schemes, but is not important here.

If a key is held down, the code is repeated at 90 ms intervals. See Figure 3. This results in a duty cycle of about 10% so that the IR LED can be pulsed at high peak power. At this duty cycle, the MLED81 can tolerate peak currents well in excess of 100 mA. Once a key is released, the MC14497 automatically sends the code for channel 62 (111110), which indicates end of transmission (EOT).
THE DECODING METHOD

Refer to Figure 4. The letters refer to time slices shown on Fig. 4. The pseudocode to decode the data looks like this:

Main:
Set up an interrupt to look for the start bit transition.

Interrupt:
Capture the timer value and save it.(a)
Look for the next transition(b), capture the timer value and subtract the saved value. This is the bit time value. Save it.
Add one quarter bit time value to the timer, and set an interrupt for when it times out.(c)
At that interrupt(c-i), look for the value, and set the carry bit accordingly.
Add the bit time to the timer, and set the interrupt. Shift the carry into the data storage location.
Have we got the six bits plus start? If not, repeat, otherwise we’re done with this word.
Is the word EOT (channel 62)? If not, the key is probably repeating. If it is, the message is complete.

THE MC68HC11 PROGRAM

This chip is easy to use because of the 16-bit add (ADDD) and subtract (SUBD) features which can used to service the timer. The MC68HC05 devices lack this, but do provide an add with carry (ADC) to implement 16-bit adds.

Port A pin 3 (PA3) is used as the input from the MC3373. This is an input capture pin (IC1). The output compare feature is required to generate the bit clock, though an output pin is not necessary.

THE MC68HC05 PROGRAM

The code for the MC68HC05 is a little different. First, since the state of the input capture pin cannot be read, the data must be routed to another pin. Therefore, PA7 is used to sample the data. The internal output compare interrupt is utilized to set the baud clock.

MC68HC11 PROGRAM LISTING

```
start     lda  #$FF
         lda  #$2
         staa TMSK2
         lda  #$1
         staa TMSK1
         staa TFLG1
         lda  #$2
         staa TCTL2
         lda  #$6
         staa count

main
  ldaa #$1
  staa TFLG2

endint
```

*Input capture interrupt service routine. The first time through, the
value of the timer is saved. the second time through, the difference
is calculated, determining the bit interval, and the time slice is
shifted by one quarter bit interval for sampling the data.

```
timeint
  brset flags,0,next
  ldd  #$1014
  std  saveit
  bsset flags,0
  bra  endint

next
  ldd  #$1014
  std  saveit
  subd saveit
  std  baud
  lsr
  lsr
  add  #$1014
  std  OUTC1
  lda  #$5FE
  anda TMSK1
  staa TMSK1
  bclr flags,0

endint
```
*occurs, sample the input line and shift the equivalent value
*into the data register. Do so for all six data bits, then word
*is complete.

```
bint
  lda PORTA
  anda #4
  rora
  rora
  rora
  rora
  lday data
  rora
  staa data
  dec count
  bne next2
  bset flags,1
  bra endin
next2
  ldd OC1
  addb baud
  std OC1
  lda #1
  staa TMSK2
  bra endint
  endin
```

MC68HC05 PROGRAM LISTING

```
start
  lda #$CO
  sta TCR
  lda #$50
  sta PADDR
  lda #6
  sta count
  lday PORTA
  lda #$0
  sta PADDR
  lda #6
  lda count
  bne next2
  lda #$7F
  sta TSR
  bclr flags,0
main
  equ *
  *main program

*input capture service routine. The first time through, the value of
*the timer is saved. The second time through, the difference is
*calculated, determining the bit interval. The time slice is shifted
*by one quarter time for sampling the data.

*timeint
  brset flags,0,next
  lday PORTA
  sta saveit
  lday ICH
  sta saveit+l
  bset flags,0
  bra endint
next
  lday ICL
  sub saveit+l
  sta byte+1
  lday ICH
  sta saveit+1
  sbdb saveit
  sta byte
  lsr
  bra
  sta OCH
  lda byte+1
  sta OCL
  lda OCL
  lsr
  sta OCH
  lda OCL
  ror
  lday #57F
  sta TSR
  bcir flags,0
  *main program
```

646
endint rti
*output compare service routine. For each interrupt that occurs,
sample the input line and shift the appropriate data bit into the
data register. Do so for all six bits, then word is complete.
*

bitint

bitint lda PADR
lsl lda data
rol sta data
dec count
beq endbit
lda OCL
add byte+1
sta OCL
lda OCH
adc byte
sta OCH
rti

endbit lda data
coma and #$5F
sta data
bset flags.1
rti
**INTRODUCTION**

One of the most powerful features shared by a wide range of Motorola MCUs is the Serial Peripheral Interface (SPI). It is primarily designed to operate as a synchronous, 8-bit communication system and is implemented entirely with on-chip hardware. This frees the CPU for other tasks and ensures a minimum of software overhead associated with the SPI system.

The SPI is available in two basic forms:
1. Level 1 SPI — implemented on the MC68HC11, HC05C4 MCUs,
2. Level 2 SPI — implemented on the MC6805S2/S3/L3/L8 MCUs.

Note that the HCMOS family of MCUs only support level 1, while level 2 is implemented only on HMOS MCUs.

Though both levels of SPI can communicate easily with each other, level 2 has a number of additional capabilities, including asynchronous communication. This application note is aimed at describing a method of achieving synchronous communication between a level 1 and level 2 SPI, and details the subtle relevant differences in the on-chip implementation of each.

**DESCRIPTION**

The two MCUs used in this application are the high performance MC68HC11 and the low cost MC6805L3.

Data is transferred between the MCUs on a single bi-directional line, with the clock supplied on an additional line. Also, to ensure initial synchronisation between each MCU, a software handshake sequence is implemented on the same lines which provide the clock and data. This has the considerable advantage of minimising the number of lines between each MCU as additional control lines are not needed.

The handshake sequence is also necessary for two other reasons; the 6805L3 receive data register is unbuffered, and it is not possible for the 6805L3 to stop transmission of data from the 68HC11 by inhibiting the clock signal. The fact that the 6805L3 data register is unbuffered means that, if a handshake sequence was not implemented, new data could begin to get clocked in before the previous data were read. Also, the on-chip configuration of the 68HC11's SPI means that, if an attempt were made to slow down or stop its clock during transfer of a byte, there would be a resultant loss of synchronism between the transmitting and receiving MCUs.

The 68HC11 software is implemented as the clock master (i.e. it provides the clock output), while the 6805L3 is the clock slave. As there are no other clock masters or slaves in the system, software is kept to an absolute minimum. In fact the main transfer routine (XFER) for the 68HC11 is only 27 bytes long, while the 6805L3 uses only 30 bytes.

The other significant advantage of this implementation is that none of the 6805L3 timers are required for SPI operation, thus ensuring a minimal impact on any other application dependent tasks the MCU may be executing.

![Figure 1. Hardware Implementation](image-url)
Figure 2. 68HC11-6805L3 SPI Timing

1. Master releases clock line by disabling SPI.
2. Slave clears data line by forcing o/p clamp on PD3.
3. Master clears clock line by enabling SPI.
4. Once clock line goes low, slave stores data in SPI data register, sets its data DDR to correct state, and enables SPI.
5. Slave releases data clamp.
7. Both master and slave detect end of transmission and read data from SPI register.
8. Slave disables SPI to ensure data line is released.
Configuring the MCUs
The method of configuring each MCU for bi-directional data transfer is slightly different, due to the differences in their SPI silicon implementation. Figure 1 shows the hardware implementation. On the 68HC11, the input and output pins must be connected together externally. On the 6805L3, this can be done internally by software, thus requiring only 1 external data I/O pin. The other 6805L3 SPI data pin is now free to be used in any other way.

As the slave select (SS) pins are unused by either MCU, they must be configured as outputs to prevent SPI fault conditions occurring.

The spare 6805L3 data pin (PD3) is used to control the bi-directional data line, thus providing a handshake signal to the 68HC11. The 68HC11’s handshake is on the clock line, and is controlled by disabling and enabling its SPI.

Data transfer and timing
Figure 2 shows details of the handshake sequence. The significant point to note from Figure 2 is that the handshake sequence is implemented purely in software, while the 8-bit data and clocks are generated by the SPI hardware.

To prevent data contention, both during data transfer and during the handshake sequence, both SPIs must operate in wired-or (open drain) mode. On the 6805L3, this is done by setting bit 3 in the miscellaneous register, while on the 68HC11, bit 5 of the SPI control register must be set. The SPI utilised on the 68HC05 family of MCUs does not support this open-drain option and cannot, therefore, be used in this application.

The clock format is: idle low, data output on positive edge and sampled on negative edge. Both SPIs must be configurred to operate with the same clock format (see Figure 2). Eight data bits are transferred, at a maximum clock rate of 125 kHz, which is limited by the 6805L3 SPI. Data transfer is preceded by the handshake sequence, which ensures that both MCUs are in the correct state, and ready to transfer a new byte of data.

The most significant bit of data appears first, on the rising edge of the first clock. Data is latched into both SPI shift registers on the falling edge of the clock. Once the last data bit is latched, the data line is released high. This is necessary to ensure correct operation of the handshake sequence. When the 68HC11 is acting as a transmitter, this state occurs automatically — as a by product of its SPI hardware implementation. However, on completion of data transmission from the 6805L3, the LSB is permanently maintained on the data line, so its driver routine has been designed to ensure that the data line is always restored to the high state.

Software routines
A glance at the software listing (see Appendix 1) reveals that the transmit and receive routines for each MCU are essentially the same! The entry and exit conditions of each are as shown in Table 1.

On the 6805L3, the X register dictates the operating mode of the transfer routine. This is necessary because the same I/O pin is used for transmitting and receiving data, so its data direction register must be changed appropriately (by the contents of X).

On the 68HC11, separate pins are used for transmitting and receiving data, so their DDR pins are set up once only in the initialisation routine.

There are other important subtle differences. Prior to reception of data, the 6805L3 SPI data register content is irrelevant, while the 68HC11 SPI data register must be loaded with $FF, to prevent data bus contention. This could occur with the 68HC11, in this application, because data is simultaneously output to and read back on the same external line

<table>
<thead>
<tr>
<th>68HC11:</th>
<th>Transmit</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACCA</td>
<td>ACCA</td>
</tr>
<tr>
<td>Entry</td>
<td>Data to send</td>
<td>$FF</td>
</tr>
<tr>
<td>Exit</td>
<td>Data sent</td>
<td>Data received</td>
</tr>
</tbody>
</table>

All other registers are unused by the 68HC11 transfer routine.

<table>
<thead>
<tr>
<th>6805L3:</th>
<th>Transmit</th>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACC</td>
<td>X Reg</td>
</tr>
<tr>
<td>Entry</td>
<td>Data to send</td>
<td>$5</td>
</tr>
<tr>
<td>Exit</td>
<td>Data sent</td>
<td>$D</td>
</tr>
</tbody>
</table>

651
during data transfer. Loading $FF$ into the 68HC11 data register ensures that this data is replaced by that transmitted from the 6805L3. 

Note that, as the 68HC11 is the clock master, it must provide the clock signal, not only when transmitting data, but also when receiving data from the 6805L3. It does this by writing to its SPI data register.

Completion of data transfer is indicated by a single flag bit (SPIF). On the 6805L3, this is bit 7 of the SPI control register, while on the 68HC11, it is bit 7 of the SPI status register. This flag bit is used to indicate completion of either transmission or reception of data.

Flag clearing techniques are quite different for the 68HC11 and 6805L3. On the latter, the SPIF flag is cleared simply by writing '0' to the flag bit. On the 68HC11, a two stage operation is required to clear the SPIF flag; the SPI status register must first be read, with the flag set, followed by an access of the SPI data register.

An examination of the SPI software drivers shows that on completion of a data transmission, the SPI data register is read again. This data should be the same as that transmitted, and provides information on whether data contention or corruption occurred during transfer. This facility could be incorporated in a data validation routine to improve reliability of data transfer.

The key features in this implementation are:

1. An orderly start-up sequence to ensure the correct initial synchronisation. As the 6805L3 is the slave, its initialisation routine is not exited until it detects a low on the clock line — this will occur only when the 68HC11 gains control of the SPI. Before this happens, all I/O pins are set to inputs, so the clock line will be pulled high by the external resistor.

2. A well defined transfer protocol is used. The master device (i.e. 68HC11) must always dictate the data transfer direction and the data stream size must be specified by the currently selected transmitter, so that the receiver knows when the last byte has been sent.

The transfer protocol operates such that after initialisation, the master MCU (68HC11) transmits a control byte to the slave (6805L3). This control byte selects the subsequent data transfer direction and is either a slave listen address or a slave talk address. If it is a slave listen address, then the 6805L3 stays in receive mode. The next byte indicates the total number of bytes to be received, followed by the data stream (see Figure 3).

Once the last byte is transferred, the 6805L3 can await a new control byte, or alternatively it can return to some other task, such as processing the previously received data. Similarly, at this point, the

---

**Figure 3. Master Transmitter — Slave Receiver**
master transmitter (68HC11) can return to another task, or send a new control byte. If the control byte now sent is a slave talk address, then the 6805L3 will switch to transmit mode, and the master will switch to receive mode. The 6805L3 will send a byte count, followed by the data stream to the master (see Figure 4).

Note that, once the last byte is transferred in either direction, both processors are free to continue other tasks or attempt a new data transfer. The handshake sequence always ensures synchronisation of data transfer, independent of the response time of either MCU.

**CONCLUSION**

Potential uses for this type of data transfer are in applications which require remote interrogation of an MCU based system via a minimal number of lines, such as:

- Development and diagnosis of engine management systems
- Smart card and key card security applications
- Instrumentation and data logging equipment.

**APPENDIX**

The demonstration programs listed in the following pages simply transfer a string of characters from the 68HC11 to the 6805L3, which converts them to upper case and sends them back again.

The HC11SPI program is listed on pages 6 to 7; the L3SPI program on pages 8 to 10.

---

**Figure 4. Master Receiver — Slave Transmitter**

1. Master sends 'slave talk address' = 2
2. Slave sends byte count = 2
3. Slave sends 2 bytes of data
4. SPI goes idle
Motorola 68HC11 X-Assembler 2.10 SYs:1030.SPI HC11SPI SA 04/30/87 13:02:11

*****************************************************************************

HC11SPI 4/9/86

SPI TRANSFER WITH HANDSHAKE ON SPI DATA AND
CLOCK LINES. THIS IS NECESSARY AS HC11'S CLOCK
CANNOT BE SLOWED BY SLAVE DEVICE.

8 A 0008 PORTD EQU 8
9 A 0009 DDRD EQU 9
10 A 0028 SPCR EQU $28
11 A 0029 SPBR EQU $29
12 A 002A SPDR EQU $2A
13 A 0024 TMSC2 EQU $24
14 A 0025 TFUC2 EQU $25
15 A 0004 MISO EQU 4
16 A 0008 MOSI EQU 8
17 A 0010 SCK EQU $10
18 A 0080 SPIF EQU $80
19 A 0040 SPE EQU $40
20 A 0000 ORG $0
21 A 0000 0001 TIMCOUNT RMB 1
22 A 0001 1C MSG1 FCB 28
23 A 0002 68620D6469 FCC /BI-DIRECTIONAL DATA TRANSFER/
24 A 0000 0000 TIMCOUNT RMB 1
25 A 0001 1C MSG1 FCB 28
26 A
27 A 0000 ORG $0
28 A C000 8EC035 START LDS #$35
29 A C003 803C BSR INIT
30 A C005 18CEO001 START1 LDY #$31
31 A C009 8O08 BSR SEND
32 A C008 18CEO001 LDD #$31
33 A C00F 8D17 BSR READ
34 A C011 202F BRA START1
35 A
36 A C013 SEND EQU
37 A C013 8E01 LDAA #1
38 A C015 8D3A BSR XFER
39 A C017 18AG00 LDAA .Y
40 A C01A 16 TAB
41 A C01B 8D94 BSR XFER
42 A C01D 1808 SEND1 INY
43 A C01F 18AG00 LDAA .Y
44 A C022 8D20 BSR XFER
45 A C024 5A DECW
46 A C025 26F6 BNE SEND1
47 A C027 39 RTS
48 A
49 A C028 READ EQU
50 A C028 8E02 LDAA #2
51 A C02A 8D2F BSR XFER
52 A C02C 8EFF LDAA #$FF
53 A C02E 8D21 BSR XFER
54 A C030 16 TAB
55 A C031 18700 STA .Y
56 A C034 1828 READ1 INY
57 A C036 8EFF LDAA #$FF
58 A C038 8D17 BSR XFER

*****************************************************************************

NUMBER OF BYTES IN DATA BLOCK

Load into EVB RAM

BUFFALO USES RAM ABOVE THIS

Y POINTS TO BEGINNING OF DATA BLOCK TO TRANSFER

TRANSMIT BLOCK TO SLAVE, STARTING WITH BYTE COUNT

RE-INITIALISE POINTER TO DATA BLOCK

AND READ THE BYTES BACK.

Enter with Y pointing at byte count of data block.

Command slave to receive

Now send slave the byte count.

But 1st store it in byte counter.

Point at next byte

And send it

Until all done

Enter with Y pointing at byte count of data block.

Command slave to transmit

Now read bytes back from slave.

1st byte is byte count so

Store it in byte counter

Then store it at beginning of data block.

Bump pointer

Then continue to read

654
Motorola M68HC11 X-Assembler 2.10 SYS:1030.SPI .HC11SPI .SA 04/30/87 13:02:11

59 A C03A 18A700 STAA .Y AND STORE BYTES
60 A C039 5A DECB UNTIL ALL DONE
61 A C03E 25F4 BNE READ1
62 A C040 39 RTS
63 A
64 A C041 INIT EQU *
65 A C041 CE1000 LDX #$1000
66 A C044 8638 LDAA #$38 ENABLE OUTPUT MODE ON SS,SCK,MOSI, INPUT ON MISO
67 A C046 A009 STAA DDRD,X (WITH SS AS OUTPUT, MODF IS DISABLED.)
68 A C048 8676 LDAA #$76 ENABLE SPI AS 125kHz CLOCK MASTER, WIRED-OR MODE
69 A C04A A728 STAA SPOR,X
70 A C04C 8618 LDAA #$18 SET DATA & CLOCK OUTPUT BUFFERS TO LOGIC '1'
71 A C04E A708 STAA PORTD,X TO GENERATE ACKNOWLEDGE CLOCK.
72 A C050 39 RTS
73 A
74 A
75 A * BIDIRECTIONAL DATA TRANSFER, WITH *
76 A * HANDSHAKE ON CLOCK & DATA LINES.
77 A * ENTERED WITH DATA IN ACCA, SPI ENABLED.
78 A * CLOCK LOW, DATA LINE DETERMINED BY SLAVE DATA PIN.*
79 A *
80 A C051 XFER EQU *
81 A C051 1F804FC BRCLR PORTD,X,MSIS,.* 1ST WAIT FOR SLAVE TO RELEASE DATA LINE.
82 A C055 102840 BCLR SPOR,X,RSPE SEND ACK ON CLOCK LINE, BY DISABLING SPI
83 A C058 1E804FC BSSET PORTD,X,MSIS,* WAIT FOR SLAVE TO ACKNOWLEDGE.
84 A C05C 1C2840 BSET SPOR,X,RSPE ENABLE SPI, CLEARING CLOCK LINE.
85 A C05F 1F804FC BRCLR PORTD,X,MSIS,* WAIT FOR SLAVE TO RELEASE DATA LINE,THEN
86 A C053 A72A STAA SPOR,X START SPI TRANSACTION BY WRITING DATA.
87 A C056 1F280FC BCLR SPOR,X,RSPIF,* NOW WAIT FOR TRANSMISSION COMPLETE FLAG.
88 A C059 A62A LDAA SPOR,X CLEAR SPIF, AND READ DATA
89 A C06B 39 RTS BEFORE RETURNING.
90 A *
91 A * ORG $FFFE
92 A * FDB START
93 A *
94 A END

***** TOTAL ERRORS 0-- 0
***** TOTAL WARNINGS 0-- 0

SYMBOL TABLE LISTING

<table>
<thead>
<tr>
<th>SYMBOL NAME</th>
<th>SECT VALUE</th>
<th>SYMBOL NAME</th>
<th>SECT VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDRO</td>
<td>A 0009</td>
<td>SPCR</td>
<td>A 0028</td>
</tr>
<tr>
<td>INIT</td>
<td>A C041</td>
<td>SPDR</td>
<td>A 002A</td>
</tr>
<tr>
<td>MOSI</td>
<td>A 0004</td>
<td>SPE</td>
<td>A 0040</td>
</tr>
<tr>
<td>MOSI</td>
<td>A 0008</td>
<td>SPIF</td>
<td>A 0080</td>
</tr>
<tr>
<td>MISO</td>
<td>A 0001</td>
<td>SSPR</td>
<td>A 0029</td>
</tr>
<tr>
<td>PORTD</td>
<td>A 0008</td>
<td>START</td>
<td>A 0000</td>
</tr>
<tr>
<td>READ1</td>
<td>A 02B5</td>
<td>START1</td>
<td>A 005</td>
</tr>
<tr>
<td>READ1</td>
<td>A 023B</td>
<td>TFG2</td>
<td>A 0025</td>
</tr>
<tr>
<td>SCK</td>
<td>A 0010</td>
<td>TIMCOUN</td>
<td>A 0000</td>
</tr>
<tr>
<td>SEND</td>
<td>A 0013</td>
<td>TMSK2</td>
<td>A 0024</td>
</tr>
<tr>
<td>SEND1</td>
<td>A 0010</td>
<td>XFER</td>
<td>A 0051</td>
</tr>
</tbody>
</table>
HANDSHARED BIDIRECTIONAL DATA TRANSFER

BETWEEN 6805L3 & 68HC11 USING SPI.

HC11 IS CLOCK MASTER.

DATA LINE CLAMP ON D3

RESERVE ENOUGH SPACE FOR RECEIVED DATA.

INITIALISE SPI, AND WAIT FOR HC11 TO BECOME READY

1ST RECEIVE COMMAND BYTE

IF SLAVE REQUESTED TO LISTEN, THEN
READ MORE DATA.

IF SLAVE REQUESTED TO TALK, THEN
SEND DATA TO MASTER.

GET NEXT BYTE, WHICH IS BYTE COUNTER.

ALSO STORE VALUE AS DATA LENGTH.

NOW READ ALL REMAINING BYTES.

(CONVERT LOWER CASE TO UPPER CASE)

AND STORE IN BUFFER

UNTIL ALL DONE,

AND RETURN.
**Motorola M6805 X-Assembler**  2.00  SYS :1030, SPI  .L3SPI  .SA 09/04/86  17:52:15

```
59 A       * SEND EQU *             Get length of data and store it in byte counter.
60 A 0DA9  LDA DATLEN
61 A 0DA9  B621 STA BYTCONT
62 A 0DA8  B720
data x
63 A 0DA7  AEO5 LDX #5
64 A 0DA6  AD22 BSR XFER
65 A 0DA5  AE20 SEND1 LDX BYTCONT
66 A 0DA4  E822 LDA DATA.X
67 A 0DA3  BF20
data x
68 A 0DA2  BF0C BSR XFER
69 A 0DA1  AE05
data x
67 A 0DA0  AD22 LDX #5
68 A 0D9F  AD1A BSR XFER
69 A 0D9E  AE06 DEC BYTCONT
70 A 0D9D  BF04 BNE SEND1
71 A 0D9C  BF0C BRA START1
72 A
73 A 00BF  INIT EQU *             Send it to master until all done, and return.
74 A 00BF A618 LDA #518
75 A 00C1 B70A STA MISC
76 A 00C0 A60D LDA #5 OD
77 A 00C3 B703 STA PORTD
78 A 00C2 A609 LDA #9
79 A 00C9 B707 STA DDRD
80 A 00CB A694 LDA #54H
81 A 00CC B70E STA SPICR
82 A 00CF 020F D BRSET SCK,PORTD.*
83 A 00D2 81 RTS
84 A
85 A
86 A       * Bidirectional data transfer on SPI
87 A
88 A       * Entered with SPI disabled, data pin high
89 A
90 A       * Entry: TX mode; ACCA=DATA, X-5
91 A
92 A       * RX mode; ACCA=TX DATA, X+5
93 A
94 A
95 A 00D3  XFER EQU *             Inhibit INT2, enable PORTD open drain.
96 A 00D3  030F D BRCLR SCK, PORTD.*
97 A 00D6  1703 BCLR CLAMP, PORTD
98 A 00D8  020F BRSET SCK, PORTD.*
99 A 00DB  B70E STA SPIDAT
100 A 00D9  180F BSET SPE, SPICR
101 A 00DF BF07 STX DDRD
102 A       * Transfer starts now!
103 A 00E1 A694 LDA #54H
104 A 00E3 A60D LDX #5 OD
105 A 00E5 0F0F D BRCLR SPIF, SPICR.*
106 A 00E8 BF07 STA SPICR
107 A 00EB BF0E STA SPIDAT
108 A 00EC BF03 STX PORTD
109 A 00ED BF07 STX DDRD
110 A 00F0 81 RTS
111 A

****  TOTAL ERRORS 0-- 0
****  TOTAL WARNINGS 0-- 0
```

657
## Symbol Table Listing

<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Sect Value</th>
<th>Symbol Name</th>
<th>Sect Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTCNT</td>
<td>0030</td>
<td>SEND</td>
<td>00A9</td>
</tr>
<tr>
<td>CLAMP</td>
<td>0033</td>
<td>SEND1</td>
<td>0081</td>
</tr>
<tr>
<td>DATA</td>
<td>0022</td>
<td>SPE</td>
<td>0004</td>
</tr>
<tr>
<td>DATLEN</td>
<td>0021</td>
<td>SPICR</td>
<td>000F</td>
</tr>
<tr>
<td>DORA</td>
<td>0004</td>
<td>SPIDAT</td>
<td>000E</td>
</tr>
<tr>
<td>DDR0</td>
<td>0007</td>
<td>SPIF</td>
<td>0007</td>
</tr>
<tr>
<td>INIT</td>
<td>008F</td>
<td>SS</td>
<td>0000</td>
</tr>
<tr>
<td>MISC</td>
<td>009A</td>
<td>START</td>
<td>0000</td>
</tr>
<tr>
<td>PORTA</td>
<td>0000</td>
<td>START1</td>
<td>0083</td>
</tr>
<tr>
<td>PORTD</td>
<td>0003</td>
<td>TACR</td>
<td>0009</td>
</tr>
<tr>
<td>PRESCL</td>
<td>0010</td>
<td>TADAT</td>
<td>0008</td>
</tr>
<tr>
<td>READ</td>
<td>0091</td>
<td>TBCR</td>
<td>000D</td>
</tr>
<tr>
<td>READ1</td>
<td>0099</td>
<td>TBDAT</td>
<td>000C</td>
</tr>
<tr>
<td>SCK</td>
<td>0001</td>
<td>XFER</td>
<td>0003</td>
</tr>
<tr>
<td>SDA</td>
<td>0002</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MC68HC805B6 LOW-COST EEPROM MICROCOMPUTER PROGRAMMING MODULE

Prepared by: Bill Mathews, Product Engineer, Motorola Semiconductors Ltd, East Kilbride, Scotland

INTRODUCTION

The EEPROM feature of the MC68HC805B6 microcomputer unit (MCU) enables the user to emulate the MC68HC05B6 and MC68HC05B4 MCU devices. This application note describes one programming technique which may be used to program the MC68HC805B6 internal EEPROM, and provides a description of the programming module used in conjunction with this application note. All that is required to program the MC68HC805B6 is the programming module (which can be implemented on a single PCB), and a twin +5V+19V dc power supply.

PROGRAMMING TECHNIQUE

A multi-byte EEPROM programming technique is used to load a user program into the MC68HC805B6 EEPROM in order to emulate the MC68HC05B6 and MC68HC05B4 devices. This type of operation is accomplished via a bootstrap mode of operation. The user program contained in an external EPROM is copied into the internal EEPROM of the MC68HC805B6 device, then verified against the contents of the EPROM.

The bootstrap program follows the following sequence following reset:

1. EEPROM6 Erase.
   The 6K EEPROM (which emulates the ROM on the MC68HC05B4 and MC68HC05B6 mask devices) is erased for a nominal 100 ms (with a 4MHz crystal). This memory area is then tested for complete erase by verifying that all bytes in the EEPROM are read $FF. If any bytes have failed to erase, then the red LED will be illuminated and the bootstrap routine will cycle back and continue to erase EEPROM6 until all bytes are erased.

2. EEPROM1 Erase.
   The 256 bytes EEPROM (which emulates the 256 bytes of byte-erasureable EEPROM on the MC68HC05B6 mask device) is erased in a similar fashion to the EEPROM6, with illumination of the red LED indicating a failure to erase. Completion of this program step ensures that the security bit (which is implemented in EEPROM1) is also erased.

3. EEPROM Program.
   EEPROM1 and EEPROM6 are then programmed in turn from the data contained in the EPROM, in increasing address order. Areas in the MC68HC805B6 memory map which do not include EEPROM will be skipped by the programming routine, as will $FF bytes, thus speeding the programming operation. During programming the green LED should flash at approximately 3Hz, though this frequency will increase for data equal to $FF.

4. EEPROM Program Verify
   EEPROM1 and EEPROM6 are verified against the contents of the EPROM, with any error causing illumination of the red LED. A successful programming operation will result in green LED on, red LED off. Note that the red LED may glow very dimly, this is normal.

PROGRAMMING OPERATION

To program the MC68HC805B6 MCU EEPROM, perform the following steps:

1. With power to the module removed install MCU and EEPROM devices into the programming module.
2. Place switch S1 to RESET position.
3. Apply +5V power supply to the programming module.
4. Apply +19V power supply to the programming module.
5. Place switch S1 in the RUN position.
6. Once the green LED is illuminated, place switch S1 to RESET position.
7. Remove +19V power supply from the programming module.
8. Remove +5V power supply from the programming module.

Note: To avoid possible damage to the MC68HC805B6 it is essential that power to the programming module is applied and removed in the sequence specified above.

PROGRAMMING MODULE CONSTRUCTION

Table 1 provides the parts list for construction of the module, with component tolerances generally not critical. A schematic of the circuit is included as Figure 1.

HARDWARE CONSIDERATIONS/DEBUG

Functionality of the module may be adversely affected by faults in the following areas:

1. IRQ and RESET
   The circuitry on these pins ensures that the IRQ pin sees a voltage level equal to approximately +10V dc during the RESET period. This is necessary for bootstrap mode capture. Correct operation of D1 and D2 should be verified.

2. VPP6
   The input to this pin is controlled via a switch driven by port pin PC7. This method ensures that a minimum +5V dc level is always present at the VPP6 pin, and that +19V dc is available during programming and erase operations on EEPROM6. Correct operation of D5, 6 and 7 should be verified. Zener diode ZD1 protects the EEPROM from...
possible damage caused by excursions greater than +20V dc.

3. VPP1

It is important that this pin (at which the output of the internal EEPROM charge pump is visible) is allowed to float. Clamping this pin to +5V dc will prevent successful programming and erase operations from taking place on EEPROM1, whilst clamping this pin to 0V could damage the device.

4. Port D

For correct bootstrap mode capture it is necessary to connect these inputs to 0V.

5. PC5/PC6

These pins provide a handshake which is not used in this application, therefore for correct operation of the module PC5 should be connected to PC6.

---

### TABLE 1. 68HC805B6 EEPROM Programming Module Parts List

<table>
<thead>
<tr>
<th>Resistors</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>100K</td>
</tr>
<tr>
<td>R2</td>
<td>1K</td>
</tr>
<tr>
<td>R3, R4</td>
<td>470</td>
</tr>
<tr>
<td>R5</td>
<td>10M</td>
</tr>
<tr>
<td>R6</td>
<td>1K</td>
</tr>
<tr>
<td>R7</td>
<td>4.7K</td>
</tr>
<tr>
<td>R8</td>
<td>1K</td>
</tr>
<tr>
<td>R9</td>
<td>4.7K</td>
</tr>
<tr>
<td>R10</td>
<td>10K</td>
</tr>
<tr>
<td>RP1-RP3</td>
<td>100K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Capacitors</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0.01uF</td>
</tr>
<tr>
<td>C2</td>
<td>10uF</td>
</tr>
<tr>
<td>C3</td>
<td>0.1uF</td>
</tr>
<tr>
<td>C4, C5</td>
<td>22pF</td>
</tr>
<tr>
<td>C6</td>
<td>100uF</td>
</tr>
<tr>
<td>C7</td>
<td>47uF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transistors</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1, Q2</td>
<td>BC337-25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Diodes</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1, D2</td>
<td>1N914</td>
</tr>
<tr>
<td>D3</td>
<td>LR3160</td>
</tr>
<tr>
<td>D4</td>
<td>LG3180</td>
</tr>
<tr>
<td>D5, D6</td>
<td>1N5822</td>
</tr>
<tr>
<td>D7</td>
<td>1N914</td>
</tr>
<tr>
<td>ZD1</td>
<td>BZY88C20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sockets</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC1</td>
<td>52 pin plcc zif</td>
</tr>
<tr>
<td>IC2</td>
<td>28 pin lif</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Connectors</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3 way terminal connector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Switches</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>2 way, toggle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR1</td>
<td>4MHz Crystal</td>
</tr>
</tbody>
</table>

---

### FIGURE 1. 68HC805B6 Parallel Bootstrap Programmer
Monitor Program for the MC68HC05B6 Microcomputer Unit

Prepared by: Bill Mathews, Product Engineer, Motorola Semiconductors Ltd, East Kilbride, Scotland

INTRODUCTION

The MC68HC05B6 HCMOS microcomputer is a member of Motorola's MC68HC05 family of low-cost single-chip microprocessors. This 8-bit microcomputer (MCU) contains an on-chip oscillator, CPU, RAM, ROM, EEPROM, A/D, PULSE LENGTH Modulated outputs, I/O, a serial Communications Interface, Timer system and Watchdog timer.

A monitor program is available in the mask ROM of a 68HC05B6, (XC68HC05B6FN MONITOR), which when used with a monitor circuit module, a power supply, and a video terminal will allow the user to write and debug small portions of 68HC05B6 code. This application note contains a description of the facilities available via the monitor software, a diagram of the monitor circuit, and a listing of the monitor code.

HARDWARE

The monitor module requires a single (+5V) power supply, and all communications between the device and terminal take place via an RS-232 link. All 68HC05B6 I/O pins are available to the user to be configured as required.

Terminal setup is as follows:
- 9600 baud, Half Duplex,
- and either 7 bit data and parity '0'
- or 8 bit data and no parity.

A circuit diagram of the monitor is given in Figure 1.

MONITOR OPERATION

The following sequence of operations should be followed:

1. Remove power supply from module
2. Insert XC68HC05B6FN MONITOR device
3. Place S1 in 'RESET' position
4. Apply +5V dc power supply
5. Connect video terminal
6. Place S1 in 'RUN' position

The 68HC05B6 will then begin to execute the monitor program, and the following message should appear on the monitor:

"Hi! I'm the MC 68HC05 B6 from MOTOROLA European Design Centre, Geneva."

A prompt "=" will be displayed to indicate that the device is ready to receive commands from the terminal. If no message appears, then the setup of the terminal should be verified.

MONITOR COMMANDS

The following commands are available:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Display the content of the registers in format.</td>
</tr>
<tr>
<td>HINZC</td>
<td>Condition code register</td>
</tr>
<tr>
<td>AA</td>
<td>Contents of Accumulator</td>
</tr>
<tr>
<td>XX</td>
<td>Contents of Index Register</td>
</tr>
<tr>
<td>PPPP</td>
<td>Program Counter</td>
</tr>
<tr>
<td>ZZ,DD</td>
<td>User specified byte (which addresses F00 to FFF) and contents. Set up using the 'V' command. Reset initialises ZZ to 008 (A/D data register).</td>
</tr>
</tbody>
</table>

Note that this command assumes that the stack pointer is at address $FA.

A

Display/Change the Accumulator

The contents of the Accumulator are displayed, then the monitor waits for input of two hex digits (new accumulator contents). Typing a carriage return will return the program to the command mode with the contents of the Accumulator unchanged.

Note that this command assumes that the Stack Pointer is at address $FA.

X

Display/Change the Index Register

The contents of the Index Register are displayed, then the monitor waits for the input of two hex digits (new Accumulator contents). Typing a carriage return will return
the program to the command mode with the contents of the Index Register unchanged.

Note that this command assumes that the Stack Pointer is at address $FA.

M nnnn Examine/Change Memory

The contents of any address in the range $000 to $1FF (Register, RAM and EEPROM1) are displayed, and the program will await further input, namely:

^ Redisplay the contents of the current address

<CR> Display the contents of the previous address (nnnn-1)

^ Open the next address (nnnn+1)

+ Increase the contents of the open location by 1

- Decrease the contents of the open location by 1

/D Replace the contents of the open location with the Ascii code for alphanumeric character D, and go to the next address.

nn Replace the contents of the currently open address with two hex digits nn and go to the next address.

Typing any other character will return the monitor to the command mode.

L nnnn List a block of memory starting at address nnnn. The default address (if nnnn is not specified) is $100. The data is displayed on screen as four blocks of eight by eight bytes, with the address printed at every sixteenth byte.

V nn Change the address of the page zero byte displayed with the R command with the hex byte specified by nn.

K nn Set Program and Erase times for operations on EEPROM1 where nn is milliseconds entered in decimal. The default values are 10ms. Typing 'enter' will skip the command.

P nn Program the entire EEPROM1 array (except address $100) with nn. This command may take some time to execute as firstly the entire array is erased, then each byte is programmed in turn.

If non-hex data is entered, then the following commands are available:

Z Program the entire array (except address $100) with Data = Address, i.e.

Address $100 = not programmed
Address $101 = $01
Address $102 = $02
Address $103 = $03 ...etc.

P Program a chequer-board pattern

Q Program an inverse chequer-board pattern

Any other character will exit this command.

E nnnn Start execution at address nnnn

C Continue program execution according to the current program counter, accumulator, index register and condition code register stored on the stack.

BREAKPOINTS AND INTERRUPTS

The SWI instruction may be used as a breakpoint. To continue following a breakpoint first replace the SWI with another command (such as NOP) then type 'C' to continue.

The interrupt vectors point to the RAM as shown below, and are spaced three bytes apart allowing the use of a JMP or BRA instruction to a service routine located in either RAM or EEPROM1.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCI</td>
<td>$00DF</td>
</tr>
<tr>
<td>Timer Overflow</td>
<td>$00E2</td>
</tr>
<tr>
<td>Timer O/P CMP</td>
<td>$00E5</td>
</tr>
<tr>
<td>Timer I/P CAP</td>
<td>$00EB</td>
</tr>
<tr>
<td>IRQ</td>
<td>$00EB</td>
</tr>
<tr>
<td>SWI</td>
<td>$08A8</td>
</tr>
<tr>
<td>RESET</td>
<td>$0C22</td>
</tr>
</tbody>
</table>

Pointing to monitor for breakpoint

Start of monitor code
I/O and INTERNAL registers definition

I/O registers

PORTA EQU $00 port A.
PORTB EQU $01 port B.
PORTC EQU $02 port C.
PORTD EQU $03 port D.
DDRA EQU $04 port A DDR.
DDRB EQU $05 port B DDR.
DDRC EQU $06 port C DDR.

EEPROM register

EECONT EQU $07 EEPROM control register.
.E1PGM EQU 0
.E1LAT EQU 1
.E1ERA EQU 2

A/D registers

ADDATA EQU $08 A/D data register.
ADSTCT EQU $09 A/D status and control register.
.COCCO EQU 7 Conversion complete flag.

PLM registers

PLMA EQU $0A pulse length mod reg A.
PLMB EQU $0B pulse length mod reg B.

Miscellaneous register

MISC EQU $0C Miscellaneous register.
.WDOG EQU 0 Watchdog control bit.
.SM EQU 1 Slow Mode.

SCI registers

BAUD EQU $0D SCI baud register.
SCCR1 EQU $0E SCI control register 1.
SCCR2 EQU $0F SCI control register 2.
.SBK EQU 0 Send break bit.
SCSR EQU $10 SCI status register.
.RDRF EQU 5  Receive data register full flag.
SCDAT EQU $11 SCI data register.
*
* TIMER registers
*
TIMCTL EQU $12  Timer control register.
.TOIE EQU 5  Timer overflow interrupt enable.
.OCIE EQU 6  Timer output compares interrupt enable.
.ICIE EQU 7  Timer input captures interrupt enable.
TIMST EQU $13  Timer status register.
.OCF2 EQU 3  Timer output compare 2 flag.
.ICF2 EQU 4  Timer input capture 2 flag.
.TOFP EQU 5  Timer overflow flag.
.OCF1 EQU 6  Timer output compare 1 flag.
.ICF1 EQU 7  Timer input capture 1 flag.
TIMIC1 EQU $14  Timer input capture register 1 (16-bit).
TIMOC1 EQU $16  Timer output compare register 1 (16-bit).
TIMCTR EQU $18  Timer free running counter (16-bit).
TIMALT EQU $1A  Timer alternate counter register (16-bit).
TIMIC2 EQU $1C  Timer input capture register 2 (16-bit).
TIMOC2 EQU $1E  Timer output compare register 2 (16-bit).
*
* MEMORY MAP DEFINITION
*
*
TEST EQU $20  TEST register
ROM0 EQU $0020  Start address of ROM0.
RAM EQU $0050  Start address of RAM.
EEPROM EQU $0100  Start address of EEPROM 256 bytes. NOT USE IN B4
.SEC EQU 0  Security bit.
ROM1 EQU $0200  Start address of ROM1.
ROMIND EQU $02BF  End address of ROM1.
ROM2 EQU $0800  Start address of ROM2. $0F00 IN B4
ROMBOT EQU $1F00  Start address of bootstrap ROM2.
*
* Miscellaneous definitions and equates
*
*
RAM1 EQU RAM+1  Working memory.
RAMINT EQU RAM+10  RAM location used for interrupt test.
VECT EQU $1FE2  Start of bootstrap vectors.
*
* Synthetized instructions
*
EOR EQU $D8  Exclusive or, 2 bytes indexed.
STA EQU $C7  Store A, extended mode.

**********************************************************************
* * * * *
* M0NIT0R P R O G R A M
* * *
Rev 1987/07/22
Programmer: O. Pilloud MOT GVA

THE MONITOR HAS THE FOLLOWING COMMANDS:

R DISPLAY OF REGISTERS HINZC AA XX PPPP VV = DD
WHERE DD IS THE CONTENT OF ANY DIRECT PAGE ADDRESS VV

A DISPLAY AND CHANGE ACCA

X DISPLAY AND CHANGE INDEX

M.... MEMORY EXAMINE/CHANGE
+ INCREMENT DATA BY 1
- DECREMENT DATA BY 1
. RE-READ SAME ADDRESS
- READ PREVIOUS ADDRESS
CR READ NEXT ADDRESS
DD CHANGE DATA
/D CHANGE DATA WITH ASCII VALUE OF D
ANYTHING ELSE EXITS MEMORY COMMAND

C CONTINUE EXECUTION

E.... EXECUTE

L.... LIST A 16 BY 16 BLOCK OF MEMORY
IF NO ADDRESS SUPPLIED, EEPROM1 IS LISTED

V.. CHANGE ADDRESS OF MEMORY DISPLAYED WITH R COMMAND

K SET WRITE AND ERASE CONSTANTS IN ms

P.. PRESET EEPROM1 WITH SUPPLIED DATA, EXCEPT ADDR $100
IF NON HEX DATA, THE FOLLOWING CAN OCCUR:

Z PRESETS DATA = ADDRESS
P CHECKER BOARD PATTERN (STARTING WITH 55)
Q CHECKER BOARD PATTERN (STARTING WITH AA)

FOR THE 3 CASES ABOVE, NO ERASE IS PERFORMED

MEMORY MAP DEFINITIONS
* Note: TEST is a write only register and its access is
* authorized only when E1LAT is cleared. When E1LAT
* is set, address $20 (POEEP6) is accessed for writing.

* DEFINITION OF MONITOR VALUES

* STACK EQU $FA

* STACK FOR MONIT

* Miscellaneous definitions and equates

** HI EQU 0
** LO EQU 1
** LONG EQU $C4
** SHORT EQU $14
** RED EQU PLMA
** GREEN EQU PLMB
** E1BW EQU 7
** E6PGM EQU 4
** E6LAT EQU 5
** E6ERA EQU 6
** TDRE EQU 7
** MBIT EQU 4
** ADON EQU 5

* MONITOR DEFINITIONS

* MPRT EQU '.
* EPRT EQU '*
* FWD EQU $0D
* BACK EQU '-
* SAME EQU '.
* PLUS EQU '+'
* MINUS EQU '-
* ASCII EQU '/
* SPACE EQU $20
* CR EQU $0D
* LF EQU $0A
* BEEP EQU $07
* EOT EQU $04

* MONITOR AND COMMON RAM DEFINITIONS

* ORG BINBCD
* CHAR RMB 1 CURRENT INPUT/OUTPUT CHARACTER
* XTEMP RMB 1 TEMP FOR GETC,PUTC
* ATEMP RMB 1 TEMP FOR GETC,PUTC
* MEMADD RMB 1 MEMORY ADDRESS FOR REGS
* FLAG RMB 1 ITEM COUNTER & TEMP FOR EEPROM R/W
* GET RMB 4 FOR PICK AND DROP SUBROUTINES
* WRITEK RMB 1 CONSTANT FOR EEPROM WRITE TIME
* ERASEK RMB 1 CONSTANT FOR EEPROM ERASE TIME
* ASC RMB 1 / FLAG

667
PAGE

* M O N I T O R  P R O G R A M *

* ORG  ROM0

* FCC /Rev /
  FCB  REV
  FCC  /.
  FCB  REL
  FCC  /

* ONE ROUTINE IN PAGE 0 - JUST FOR THE (C)HECK OF IT

* PCC --- PRINT CONDITION CODE REGISTER

PCC   LDA  STACK+1  GET CCR
       ASLA   MOVE H BIT TO BIT 7
       ASLA
       ASLA
       STA  GET   SAVE IT
       CLRX

PCC2  LDA  ".
       ASL  GET   PUT BIT IN CARRY BIT
       BCC  PCC3  BIT OFF MEANS PRINT "."
       LDA  CCSTR,X  PICKUP APPROPRIATE CHAR

PCC3  JSR  PUTC  PRINT "." OR CHAR
       INCX  POINT TO NEXT IN STRING
       CPX  #5  5 BITS ARE GOOD ENOUGH
       BLO  PCC2
       RTS

* Fill page 0 ROM

  FCC  /0123456789AB/

***************************************************************************

PAGE

* ORG  ROM2

* NOW, OTHER ROUTINES IN MAIN EEPROM

* TABLES

BBTBL  FCB  $A,$14,$28,$50

*
CCSTR FCC /HINZC/
* 
MSG FCC /Hi! I'm the MC 68HC05 B6 /
FCC /From MOTOROLA European /
FCC /Design Center, Geneva/
FCB EOT
* 
CMA EQU * COMMON MESSAGE AREA
NGM FCB '?,BEEP,CR,LF,EOT
ERA FCC /Erase: /
FCB EOT
WRI FCC /Write: /
FCB EOT
MS FCC /ms /
FCB EOT
* 
* 
* SETA --- EXAMINE / CHANGE ACCUMULATOR
* 
SETA LDX #STACK+2 POINT TO A
BRA SETANY
* 
* SETX --- EXAMINE / CHANGE INDEX
* 
SETX LDX #STACK+3 POINT TO X
* 
* SETANY - PRINT $X$ AND CHANGE IF REQUESTED
* 
SETANY LDA ,X PICK UP THE DATA
JSR PUTBYT AND PRINT IT
JSR PUTC AND A SPACE
JSR GETBYT CHANGE ?
BCS BEEP ERROR, NO CHANGE
STA ,X ELSE REPLACE WITH NEW VALUE
BRA MAIN NOW RETURN
* 
* REGS --- PRINT CPU REGISTERS
* 
REGS JSR PCC PRINT CCR
JSR PUTC SEPARATE FROM NEXT STUFF
CLR GET+1+HI POINT TO PAGE ZERO
LDA #STACK+2
STA GET+1+LO POINT TO A ON STACK
JSR OUT2HS NOW PRINT A
JSR OUT2HS AND X,
JSR OUT4HS THE PC,
JSR DISADD THE CURRENT ADDRESS
LDA \'=
JSR PUTC PRINT \'=
JSR PUTC SPACE
CLR GET+1+HI CLR ADDRESS CARRY
JSR PRDAT FINALLY THE DATA AND A SPACE
* 
FALL INTO MAIN LOOP
* 
* MAIN --- PROMPT, GET AND DECODE COMMAND
```
* MAIN JSR CRLF START A NEW LINE
LDA #MPRT
JSR PUTC PRINT THE PROMPT
JSR GETC GET COMMAND
JSR PUTS PRINT SPACE
*
CMP '#A CHANGE A
BEQ SETA
CMP '#X CHANGE X
BEQ SETX
CMP '#R REGISTERS
BEQ REGS
CMP '#E EXECUTE
BEQ EXEC
CMP '#C CONTINUE
BEQ CONT
CMP '#M MEMORY
BEQ MEMORY
CMP '#L LIST BLOCK OF MEM
BEQ LISTR
CMP '#V ADDRESS CHANGE
BEQ MEMDIS
CMP '#P PRESET EEPROM
BEQ PRESTR
CMP '#K SET CONSTANTS
BEQ CONSTR
FALL INTO BLEEP
*
* BLEEP -- PRINT '?' AND PROTEST
*
BLEEP LDX #NGM-CMA ?+BEEP MESSAGE
JSR PUTMSG
BRA MAIN
*
*
LISTR JMP LIST
PRESTR JMP PRESET
CONSTR JMP CONSTR
*
* EXEC --- EXECUTE FROM GIVEN ADDRESS
*
EXEC JSR GETBYT GET HIGH BYTE
BCS BLEEP NOT A HEX DIGIT
AND #$1F MAX ADDR $1FF
TAX SAVE FOR A SECOND
JSR GETBYT NOW THE LOW BYTE
BCS BLEEP BAD ADDRESS
STA STACK+5 PC LOW
STX STACK+4 PC HIGH
*
* CONT --- CONTINUE USER PROGRAM
*
CONT RTI COULDN'T BE SIMPLER
```
* MEMDIS - MEMORY ADDRESS CHANGE (FOR REGISTER DISPLAY) *

MEMDIS JSR  DISADD  PRINT PRESENT ADDRESS
JSR  GETBYT  GET NEW VALUE
BCS  BLEEP  RETURN FOR NON VALID INPUT
STA  MEMADD
BRA  MAIN  RETURN

* MEMORY - MEMORY EXAMINE / CHANGE *

MEMORY JSR  MEM8  GET ADDRESS
BCS  BLEEP  NOT HEX CHAR
MEM2 JSR  CRLF  BEGIN NEW LINE
JSR  PRADD  PRINT CURRENT ADDRESS
JSR  PRDAT  AND ASSOCIATED DATA
JSR  GETBYT  TRY TO GET A BYTE
BCS  MEM3  MIGHT BE A SPECIAL CHAR
MEMA JSR  DROP  OTHERWISE, PUT IT AND CONTINUE
MEM4 JSR  BUMP  GOTO NEXT ADDRESS
BRA  MEM2  AND REPEAT

MEM3 CMP  #SAME  RE-EXAMINE SAME ?
BEQ  MEM2  YES, RETURN WITHOUT BUMPING
CMP  #FWD  GO TO NEXT ?
BEQ  MEM4  YES, BUMP THEN LOOP
CMP  #BACK  GO BACK ONE BYTE ?
BEQ  MEM5  YES, GO DECREMENT ADDRESS
CMP  #ASCII  NEXT VALUE ASCII ?
BEQ  MEM9  GO READ VALUE
CMP  #PLUS  INCREMENT DATA ?
BEQ  MEM6  YES, GO READ DATA
CMP  #MINUS  DECREMENT DATA ?
BNE  BLEEP  NO, EXIT MEMORY COMMAND

JSR  PICK  GET THE DATA BYTE
DECA
BRA  MEM7  AND GO PUT IT BACK
MEM6 JSR  PICK  GET THE DATA BYTE
INCA
MEM7 JSR  DROP  PUT IT BACK
BRA  MEM2  READY

MEM5 DEC  GET+1+LO  DECREMENT LOW BYTE
LDA  GET+1+LO  CHECK FOR UNDERFLOW
CMP  #$FF
BNE  MEM2  NO UNDERFLOW
DEC  GET+1+HI
LDA  GET+1+HI  SAME FOR HIGH BYTE
CMP  #$FF
BNE  MEM2  OK
LDA  #$1F  HIGHEST ADDRESS IS $1FF
STA  GET+1+HI
BRA  MEM2

MEM9 JSR  GETC  READ 1 BYTE
* BRA MEMA

* MEM8 JSR GETBYTE BUILD ADDRESS
BCS MEND NOT HEX CHAR
STA GET+1+HI
JSR GETBYTE
BCS MEND
STA GET+1+LO ADDRESS IS NOW COMPLETE
MEND RTS AND IN GET+1 HI & LO

* BULKW -- BULK WRITE EEPROM1 - DATA IS IN A - A IS UNCHANGED
* BULK OPERATION BEING DISABLED IN USER MODE, THE
* BULK OPERATIONS ARE PERFORMED BY SUCCESSIVE
* BYTE OPERATIONS.

BULKW LDX #01
STX GET+1+LO
STX GET+1+HI SET UP ADDRESS $101
NXTBTR BSR BYTEW BYTE WRITE
JSR BUMP
BRCLR 1,GET+1+HI,NXTBTR LOOP
RTS

* BULKE -- BULK ERASE OF EEPROM1 - A IS UNCHANGED
* SEE ABOVE ABOUT BULK OPS

BULKE CLR GET+1+LO
LDX #01 SET UP ADDRESS
STX GET+1+HI
NXTBTE BSR BYTEE BYTE ERASE
JSR BUMP
BRCLR 1,GET+1+HI,NXTBTE LOOP
RTS

* PRESET -- EEPROM SET OR ERASE

PRESET JSR GETBYTE GET DATA BYTE
BCS BLEEPZ MAY BE SPECIAL COMMAND
BSR BULKE ERASE E2PROM
BSR BULKW WRITE E2PROM
MAINT JMP MAIN RETURN

* BLEEPZ CMP #'Z DATA = ADDRESS ?
BEQ DADD
CMP #'P CHKBOARD 5'S ?
BEQ PS5
CMP #'Q CHKBOARD A'S ?
BEQ PSA ELSE FALL INTO BLEEP

* BLEEPK LDA SCDAT CLEAR RX STATUS
BLEEPR JMP BLEEP

* DADD --- PRESET EEPROM WITH DATA = ADDRESS

672
(ADDRESS $100 UNCHANGED)

DADD  LDA  #$01    BOTTOM OF EEPROM
       STA  GET+1+HI
DADLP STA  GET+1+LO ADDRESS LSB
BSR   BYTEW WRITE DATA IN A
INCA  NEXT ADDRESS, NEXT DATA
BNE   DADLP NO OVERFLOW YET
DADND BRA  MAINT JOB DONE

* *
* CHECKBOARD PATTERNS*
* *

PS5  LDA  #$55 START VALUE
BRA  PSC
PSA  LDA  #$AA
* *
PSC  LDX  #01
STX  GET+1+LO
STX  GET+1+HI
* *
AX25 LDA  #$0F
AND  GET+1+LO
BNE  TNC
COM  ATEMP
* *
TNC  LDA  ATEMP
JSR  BYTEW
JSR  BUMP
BRCLR 1,GET+1+HI,AX25
BRA  DADND

* *
* BYTEW -- BYTE WRITE TO EEPROM1 - DATA IS IN A
ADDRESS IN GET+1&2 - A IS UNCHANGED *
* *
BYTEW STA  FLAG SAVE A
BSET  .E1LAT,EECONT PROG LATCH ENABLE
JSR  DROP1 PUT ADDRESS + DATA
BSET  .E1PGM,EECONT
LDA  WRITEK GET WRITE TIMING CONSTANT
JSR  ADJDEL
BCLR  .E1LAT,EECONT END OP
LDA  FLAG  RESTORE A
RTS

* BYTER -- BYTE ERASE - A IS UNCHANGED AND ADDRESS
IS IN GET+1 HI & LO *
* *
BYTER STA  FLAG  SAVE DATA
BSET  .E1LAT,EECONT PROG LATCH ENABLE
BSET  .E1ERA,EECONT
JSR  DROP1 PUT ADDRESS
BSET .E1PGM,EECONT
LDA ERASEK GET ERASE TIMING CONSTANT
JSR ADJDEL
BCLR .E1LAT,EECONT
LDA FLAG RESTORE A
RTS

* LIST --- LIST 4 BLOCKS OF 64 BYTES ON ONE PAGE *

LIST JSR MEM8 GET START ADDRESS
BCC GOL GOOD ADDRESS
LDA #01
STA GET+1+HI SET EEPROM1 ADDRESS
CLR GET+1+LO

GOL JSR CRLF
CLRX COUNTER 256 BYTES
LL1 JSR PRADD PRINT ADDRESS
JSR PUTS AND A SPACE
LNEXT JSR PRDAT PRINT DATA
JSR BUMP NEXT BYTE
STX XTEMP A GOOD THOUGHT FOR
JSR ADJ10 UNBUFFERED TERMINALS
LDX XTEMP
DEX DATA COUNT DECREMENT
TXA
AND #%00001111 TEST FOR 16TH DATA BYTE
BEQ LINE
AND #%00000111 TEST FOR 8TH DATA BYTE
BNE LNEXT
JSR PUTS INSERT EXTRA 2 SPACES
JSR PUTS
BRA LNEXT
MEC9BH JMP MAIN

* LINE BSR ASCIIR PUT ASCII LINE
CPX #00
BEQ MEC9BH
JSR CRLF
CMPX #128 TEST FOR 128 BYTE PAGE
BNE NOPAGE
JSR CRLF ADD EXTRA LINE IF TRUE
NOPAGE BRA LL1

* ASCIIR LDA GET+1+LO DECREMENT MEMORY POINTER
SUB #16
STA GET+1+LO
BCC NOBORO
DEC GET+1+HI
LDA GET+1+HI
AND #$1F JUST IN CASE
STA GET+1+HI
NOBORO TXA BACK UP BYTE COUNTER
ADD #16
TAX
JSR PUTS SEPARATE WITH 2 SPACES
JSR PUTS
LPASC BSR PUTASC PUT CHAR
JSR BUMP
DEX
TXA
AND #$0F
BEQ LINER
BRA LPASC FINISH THIS LINE
LINER RTS
*
   PUTASC JSR PICK
   AND #$7F
   CMP #$20
   BHS DISP '. ' BELOW $20
   LDA '. '
   DISP CMP #$7F
   BNE DISP1
   LDA #$20 SPACE FOR DEL ($7F)
   DISP1 JSR PUTC
   RTS
*
*
   DISPLAY AND CHANGE WRITE AND ERASE CONSTANTS
   *
   CONSTS CLR FLAG 0=ERA; 1=WRI
   LDX #ERA-CMA
   BRA CM1
   CML LDX #WRI-CMA
   CM1 JSR CRLF
   JSR PUTMSG PUT EITHER WRI OR ERA
   LDA ERASEK
   BRCLR 7,FLAG,CM2
   LDA WRITEK
   CM2 BSR BINBCD
   JSR PUTBYT PRINT BCD VALUE
   JSR PUTS
   LDX #MS-CMA
   JSR PUTMSG PRINT ms
*
   JSR GETNYB GET MS NYBBLE
   BCS NWER IF ERROR
   CMP #$9
   BHI NWER BCD INPUT !
   STA GET SAVE MSD
*
   JSR GETNYB GET LS NYBBLE
   BCS NWER IF ERROR
   CMP #$9
   BHI NWER BCD INPUT !
   BSR BCDBIN SAVE IN BINARY
   TSTA CMP #$00
   BEQ NWER MIN IS 1 ms
   BRSET 7,FLAG,CM3 WRI K
   STA ERASEK
CMX DEC FLAG
BRA CML
CM3 STA WRITEK
CM4 JMP MAIN
*
NWER CMP #CR CR ?
BNE CMD
BRCLR 7,FLAG,CMX
BRA CM4
CMD JMP BLEEP
*
*
* UTILITIES
*
*
BCDBIN CLRX POINT TO CONV TABLE
BCBIL LSR GET
BCC BCBIL1
ADD BBTBL,X
BCBIL1 INX #$4
CPX BNE BCBIL
RTS A HAS BCD NB
*
*
BINBCD CLR COUNT FUTURE BCD
STA CHAR SAVE ENTRY PARAMETER
BBCLOP LDA CHAR
BEQ DONEB
DEC CHAR
INC COUNT
*
LDA COUNT OVF 9 => A
AND #$0F
CMP #$0A
BNE BBC1 IF NOT
LDA COUNT
ADD #$06 ADJUST
STA COUNT
*
BBC1 BRA BBCLOP
DONEB LDA COUNT GET BCD VALUE
RTS
*
*
* PICK --- GET BYTE FROM ANYWHERE IN MEMORY
* THIS IS A HORRIBLE ROUTINE (NOT MERELY
* SELF MODIFYING, BUT ALSO SELF CREATING)
* GET+1 HI & LO POINT TO ADDRESS TO BE READ.
* BYTE IS RETURNED IN A
* X IS UNCHANGED AT EXIT
*
676
PICK STX XTEMP SAVE X
LDX #$D6 $D6 = LDA 2-BYTE INDEXED
BRA COMMON

* DROP --- PUT BYTE TO ANY MEMORY LOCATION
* HAS THE SAME UNDESIRABLE PROPERTIES
* AS PICK
* A HAS BYTE TO STORE AND GET+1 HI & LO POINTS
* TO LOCATION TO STORE, A AND X ARE
* UNCHANGED AT EXIT
* THE FLOW IS DIFFERENT WHETHER FOR RAM OR EEPROM
*
DROP BSR TSTEE WITHIN EEPROM ?
BCC DROP1 NOT EEPROM
JSR BYTER ERASE BYTE
JSR BYTEW WRITE BYTE
RTS
DROP1 STX XTEMP SAVE X
LDX #$D7 $D7 = STA 2-BYTE INDEXED
*
*
COMMON STX GET PUT OPCODE IN PLACE
LDX #$81 $81 = RTS
STX GET+3 NOW THE RETURN
CLRX WE WANT ZERO OFFSET
JSR GET EXECUTE THIS MESS
LDX XTEMP RESTORE X
RTS AND EXIT
*
*
TSTEE -- TEST THE ADDRESS IN GET+1 HI & LO AND RETURN
* WITH CARRY SET IF IT IS A VALID EEPROM ADDRESS, AND CLEARED OTHERWISE.
*
TSTEE LDX GET+1+HI
CPX #01 TEST HI BYTE
BNE NOEE NOT EEPROM
SEC
RTS
NOEE CLC
RTS
*
*
BUMP --- ADD ONE TO CURRENT MEMORY POINTER
* A AND X UNCHANGED
*
BUMP INC GET+1+LO INCREMENT LOW BYTE
BNE BUMP2 NON-ZERO MEANS NO CARRY
INC GET+1+HI INCREMENT HIGH NYBBLE
BUMP2 RTS
*
* OUT4HS - PRINT BYTE POINTED TO AS AN ADDRESS AND
* BUMP POINTER - X IS UNCHANGED AT EXIT
*
677
OUT4HS BSR PICK GET HIGH BYTE
BSR PUTBYT AND PRINT IT
BSR BUMP GO TO NEXT ADDRESS
* FALL INTO OUT2HS
*
* OUT2HS - PRINT BYTE POINTED TO, THEN A SPACE,
* BUMP POINTER. X IS UNCHANGED AT EXIT
*
OUT2HS BSR PICK GET THE BYTE
BSR PUTBYT
BSR BUMP GO TO NEXT
BSR PUTS FINISH UP WITH A SPACE
RTS
*
* DISADD - PRINT ONE BYTE ADDRESS IN MEMADD
*
DISADD LDA MEMADD GET ADDRESS
STA GET+1+LO SET UP TO PRINT
BSR PRADD1 PRINT ADDRESS (PAGE 0)
RTS
*
* PRADD -- PRINT CURRENT ADDRESS FROM GET+1 HI & LO
*
PRADD LDA GET+1+HI PRINT CURRENT LOCATION
AND #$1F max $1FFF
STA GET+1+HI CONVENIENTLY RESTORE 1X
BSR PUTBYT
PRADD1 LDA GET+1+LO
BSR PUTBYT
BSR PUTS THEN A SPACE
RTS
*
* PRDAT -- PRINT DATA POINTED TO BY GET+1 HI & LO
*
PRDAT BSR PICK GET THAT BYTE
BSR PUTBYT PRINT IT
BSR PUTS ANOTHER SPACE
RTS
*
* PUTBYT - PRINT £A| IN HEX - A AND X UNCHANGED
*
PUTBYT STA GET SAVE A
LSRA
LSRA
LSRA
LSRA SHIFT HIGH NYBBLE DOWN
BSR PUTNYB PRINT IT
PTSN LDA GET
BSR PUTNYB PRINT LOW NYBBLE
RTS
*
* PUTNYB - PRINT LOWER NYBBLE OF A IN HEX
* A AND X UNCHANGED
* HIGH NYBBLE OF A IGNORED
*
PUTNYB STA GET+3        SAVE A IN YET ANOTHER TEMP
    AND #$0F        MASK OFF HIGH NYBBLE
    ADD #'0         ADD ASCII ZERO
    CMP #'9         CHECK FOR A-F
    BLS PUTNY2
    ADD #'A-9-1     ADJUSTMENT FOR HEX A-F
PUTNY2 JSR PUTC
    LDA GET+3      RESTORE A
    RTS

*  * CRLF --- PRINT CARRIAGE RETURN - LINE FEED
*  A AND X UNCHANGED
*  *
CRLF STA GET          SAVE A
    LDA #CR
    JSR PUTC
    LDA #LF
    JSR PUTC
    LDA GET      RESTORE A
    RTS

* PUTS --- PRINT A SPACE - A AND X UNCHANGED
*
PUTS STA GET          SAVE A
    LDA #SPACE
    JSR PUTC
    LDA GET      RESTORE A
    RTS

* GETBYT - GET A HEX BYTE FROM TERMINAL
*  *
*  A GETS THE BYTE TYPED IF IT WAS A VALID HEX
*  NUMBER, OTHERWISE A GETS THE LAST CHAR TYPED.
*  THE C-BIT IS SET ON NON HEX CHARACTERS, CLEARED
*  OTHERWISE. X IS UNCHANGED IN ANY CASE.
*  *
GETBYT BSR GETNYB     BUILD BYTE FROM 2 NYBBLES
    BCS NOBYT       NON HEX CHAR
    ASLA
    ASLA
    ASLA
    ASLA          SHIFT NYBBLE TO HIGH NYBBLE
    STA GET       SAVE IT
    BSR GETNYB    GET LOW NYBBLE NOW
    BCS NOBYT     NON HEX CHAR
    ADD GET       C-BIT CLEARED
NOBYT RTS

*  * GETNYB - GET HEX NYBBLE FROM TERMINAL
*  *
*  A GETS THE NYBBLE TYPED IF IT WAS IN THE RANGE 0-F,
*  OTHERWISE A GETS THE CHARACTER TYPED. THE C-BIT IS
*  SET ON NON HEX CHARACTERS, CLEARED OTHERWISE.
*  X IS UNCHANGED
*  *
GETNYB BSR GETC GET THE CHARACTER
STA GET+3 SAVE IT JUST IN CASE
SUB # '0' SUBTRACT ASCII ZERO
BBI NOTEXX WAS LESS THAN '0'
CMP #9
BLS GOTIT
SUB # 'A' - '9' - 1 FUNNY ADJUSTMENT
CMP #$F TOO BIG ?
BHI NOTEXX WAS GREATER THAN 'F'
CMP #9 CHECK BETWEEN ASCII 9 AND A
BLS NOTEXX
GOTIT CLC C=0 MEANS GOOD HEX CHAR
RTS
NOTEXX LDA GET+3 GET SAVED CHAR
SEC
RTS RETURN WITH 'ERROR'

* ADJDEL - DELAY FOR EEPROM ROUTINES = TO fA| ms
*
ADJ10 LDA #10
ADJDEL LDX #83 CONSTANT
ALL1 BRCLR 4,ADSTCT,*+3 DUMMY
BRCLR 4,ADSTCT,*+3 DUMMY
BRCLR 4,ADSTCT,*+3 DUMMY
BRN * DITTO
DECX
BNE ALL1
DECA
BNE ADJDEL LOOP A TIMES
RTS

* * *
* PUTMSG - PRINT THE MESSAGE POINTED TO BY X *
*
PUTMSG LDA CMA,X GET NEXT CHARACTER
CMP #EOT
BEQ NDMSG
BSR PUTC SEND CHAR
INX
BRA PUTMSG
NDMSG RTS

* * *
* SERIAL I/O ROUTINES *
*
* Initialise the SCI *
*
SCINIT BCLR MBIT,SCCR1 8 data bits
LDA #$11000000 baud rate 9600
STA BAUD
LDA #$00001100 TE / RE
STA SCCR2 end of init
STA SCSR clear TDRE & TC bits
* * * GETC : Routine GETC services the SCI, it does that by polling
* the RDRF (received data ready flag). It returns with
* the byte of data in ACCA.
* * *
GETC
BRCLR .RDRF,SCSR,* Possibly wait for char
GDATA
LDA SCDAT get data & clear RDRF
CMP #'/ NEXT CHAR ASCII ?
BNE CHARs
DEC ASC FLAG IT
RTS
CHARS
BRSET 7,ASC,SCHAR
CMP #$40 NEXT CHAR ASCII ?
BLS NOCHAR
AND #$1011111 UPPER CASE
SCHAR
CLR ASC
NOCHAR
RTS
* * *
* PUTC : Routine PUTC services the SCI. It polls the TDRE
* (Transmit Data Register Empty), and puts the char
* when true.
* *
PUTC
BRCLR TDRE,SCSR,* WAIT
STA SCDAT
RTS
* *
* ================ ENTRY ================
* * MONIT - ENTRY POINT FROM RESET *
* *
MONIT
LDA #10 10 ms BY DEFAULT
STA ERASEK SET ERASE DEFAULT TIME
STA WRITEK SET WRITE DEFAULT TIME
JSR SCINIT INIT SCI
LDA #ADDATA
STA MEMADD DISPLAY ADR BY DEFAULT
JSR CRLF START A BRAND NEW LINE
CLRx POINT TO START OF MESSAGE
BABBLE
LDA MSG,X GET NEXT CHARACTER
BRCLR 4,PORTD,BAB1 ROM MESSAGE
LDA EEPROM+1,X GET NEXT CHAR (EEPROM1 MESSAGE)
BAB1
CMP #EOT
BEQ BABND IF END OF MESSAGE
Bsr PUTC PRINT IT
INcx POINT TO NEXT CHAR
BNE BABBLE MORE !
BABND
JSR CRLF SEPERATE MESSAGE FROM COMMANDS
SWI GO TO MONITOR ROUTINES
Bra MONIT LOOP AROUND
* EEPROM1 BURN IN TEST ROUTINE.
* SET UP REQUIRED NB OF ITERATION IN $70:$71
* AND DATA TO BE PROGRAMMED IN $72.
* NOTE: MAXIMUM NB OF ITERATION IS $7FF.
*
* ABCNT EQU $70
ABDAT EQU $72
*
ABCD JSR CRLF
ABL LDA ABDAT
JSR BULK
JSR BULKW
LDA ABCNT+LO
DECA
STA ABCNT+LO
CMP #$FF
BNE NOBURO
DEC ABCNT+HI
BMI NDAB
NOBURO LDA ABCNT+HI
JSR PUTBYT
LDA ABCNT+LO
JSR PUTBYT
JSR CRLF
BRA ABL
NDAB SWI
*
* =========================================================================
* VECTORS
* 
* The unused vectors point to RAM, so as to be available
* for test purposes (RAM Bootloader, SCI loader). Their
* positioning allows 10 bytes for the stack, that is 2
* interrupt levels, or 1 interrupt and 2 subroutine levels.
* 
* FDB STACK-9-18 SCI
FDB STACK-9-15 TIM OVF
FDB STACK-9-12 TIM OUT COMP
FDB STACK-9-9 TIM IN CAP
FDB STACK-9-6 IRQ
FDB MAIN SWI
FDB MONIT RESET
*
* =========================================================================
* END
* 
END

682
An introduction to
SECURE SINGLE CHIP
MICROCOMPUTER MANUFACTURE

By Mike Paterson
Motorola Ltd.
East Kilbride

INTRODUCTION

Motorola is one of the world leaders in the design and manufacture of advanced semiconductor devices. We have a major manufacturing capability at East Kilbride, in Scotland’s “Silicon Glen”, where we manufacture many of our latest microprocessor products—primarily for the European market, though we ship product world-wide, including to Japan and the U.S.A. Our business is in selling silicon; within this overall goal however we have, for more than a decade, worked on developing the necessary technology, design and manufacturing techniques required to produce a range of secure microcomputer products specifically for the SmartCard marketplace.

One of the most important and fundamental issues for SmartCards is security. There are many financial, commercial, industrial and even military applications for SmartCards which are viable only if they can provide the appropriate levels of security demanded by such applications. This brings us to the title of this Engineering Bulletin, does it mean “manufacture of secure microcomputers” or “secure manufacture of microcomputers”? Motorola is perhaps unique among today’s silicon suppliers in that it can provide microcomputers both designed from the outset to be secure and from a secure manufacturing line. We recognise that the overall security of any application is dependent not only on the intrinsic security of the device itself and its ability to prevent unauthorised access, but also on the measures taken by the semiconductor supplier to prevent fraudulent tampering with the device during and after manufacture. The customer’s application software is also a critical link in the security chain. If the semiconductor manufacturer wants to play an active part in the SmartCard marketplace, he must recognise security as a key parameter in everything from the conceptual design of a new microcomputer, through his manufacturing process, even to his delivery of the product to his customer.

Security in relation to microcomputers used for SmartCards can be grouped into three main categories:

1. Designed-in (“Intrinsic”) security
2. Manufacturing security
3. Application security

Clearly Design and Manufacturing Security are the responsibility of the semiconductor manufacturer, and are the main subjects of this Engineering Bulletin. Application Security on the other hand is ultimately the responsibility of the SmartCard systems designer. His task is to design the (secure) application software required to meet the system specification of, for example, a financial transaction or an industrial security application. Application security takes into consideration the design of the chip and the on-chip security features which can be utilised by the user’s application software. But even here the semiconductor manufacturer has a role to play; the manufacturer needs to be able to provide Applications Engineers well versed in the hardware and software features of his microcomputers and of their development support tools, to act as consultants in helping customers develop their own hardware and software.
This Engineering Bulletin will start by defining what we mean by a single chip microcomputer and what is fundamentally different about a secure one; then it will look briefly at the history of Motorola's involvement in this business, and how it has developed its SmartCard product line capability over the last decade to where it is today. Some of the techniques available to the design engineer which allow him to include security features in the device itself will be explored, along with the constraints placed on him by the demands of the SmartCard market and the need to test these devices. It will look very briefly at the wafer fabrication process and what security measures can and must be taken. Finally, it will consider the device testing process, and the particular problems associated with the testing of secure microcomputers.

SINGLE CHIP MICROCOMPUTERS

What do we mean by a single chip microcomputer, and what is special about a secure one? A single chip microcomputer (commonly referred to as an MCU) is a full microprocessor system integrated on to a single piece of silicon. It is a complete computer system in miniature, containing almost all the resources required to implement a particular application, or range of applications. In addition to the Central Processing Unit (CPU) and its control circuitry, it typically contains blocks of different types of memory, and a selection of hardware functions, optimised for general or sometimes very specific application areas. The only computer-like resource it lacks are the external human or machine interface devices such as keyboards, displays, disk drives, transducers and sensors. Most MCUs contain, at the very least, areas of random access memory (RAM) and read only memory (ROM). The RAM is used by the CPU for temporary data storage during calculations and transactions. Data can be read from or written to the RAM by the CPU in the space of a few microseconds, however any data in the RAM is lost when the electrical power supply is removed. The ROM area is used to store information which will not change throughout the working life of the MCU. Primarily it is used to store the customer’s application software (known as the user software or “ROM code”). This controls the sequence of logical operation and decision making of the CPU. It can also contain any fixed data, in the form of look-up tables for example, which may be required by the application. All this information is built into the ROM during the silicon manufacturing process and can never be altered after the manufacturing process is complete. MCUs incorporating arrays of non-volatile memory, such as EPROM (erased by UV light) and EEPROM (electrically erasable) are available for applications where variable data has to be kept for long periods of time, even when power is removed from the MCU. This is particularly important in SmartCard applications where transaction data and other records must be updated every time the card is used (possibly several times a day depending on the type of application) and which must be retained for weeks, months, or even years in some cases. Typical on-chip hardware functions for general applications are simple counter/timers, byte-wide I/O ports and serial communications interfaces.

An important thing to realise is that most typical single chip microcomputers are designed to function in a number of different operating modes which can be selected by the user. In the single chip user-mode, which is the normal mode of operation in most applications, the CPU runs under the control of the user software built into the on-board ROM. Some MCUs support expanded-mode operation where internal data and address buses are connected to the I/O pins to allow the CPU to access additional memory and I/O outside the MCU. Other operating modes are provided for testability.
A secure single-chip microcomputer can include any or all of the features just described, but it also has the built-in capability to prevent, by various means, unauthorised access to the CPU, the memory arrays, the user/application software, and any data being processed or stored within the device, at any time. After it has been tested and passed as fully functional by the semiconductor manufacturer, the only possible mode of operation for a secure microcomputer must be the user-mode, i.e. under the complete control of the user software in the on-board ROM.

HISTORY

In 1977 Motorola (working with one of its European customers) began a feasibility study concerned with putting a microcomputer and a non-volatile memory within an ISO7810 credit card. This solution was fully functional by 1979. As a result of evaluation and feasibility studies it soon became apparent that "multi-chip computer" solutions have a number of inherent disadvantages which can detract significantly from their suitability for secure, reliable, high-volume SmartCard applications. In multi-chip solutions, the interconnections forming the control and data buses between the chips are easily accessible from the outside world and the data being transmitted across them may be intercepted and monitored. This can seriously limit the inherent security of such a multi-chip solution. In single chip solutions however the interconnections are buried deep within the structure of the silicon die and, in devices intended for SmartCard applications, the silicon designer can use a variety of techniques to ensure that these interconnections cannot be accessed from the outside world. Multi-chip solutions are also inherently less reliable and usually more expensive than single chip solutions because physical stresses can damage the external interconnections and fabrication, test and assembly costs have to be incurred for each chip. We believe therefore that the concept of a single-chip microcomputer solution is important to the successful realisation of secure, reliable, high-volume SmartCard applications.

Work began in Motorola's European Design Office in Geneva in 1980 on the design of a SmartCard chip to a general specification agreed with a potential customer. The design was based upon the CPU of the highly successful M6805 single-chip MCU, with 8 K bits of UV erasable memory (EPROM), as well as RAM and ROM. This was all integrated on a single NMOS technology silicon chip of less than 20 mm². The merging of EPROM and NMOS MCU technologies on one piece of silicon was a new area for Motorola and our Advanced Products Research and Development Laboratory (APRDL) was given the task of designing and developing a completely new wafer fabrication process to support this combined technology. First silicon was successfully produced in APRDL in 1981, and in 1982 the new production process was transferred from APRDL to the volume wafer fabrication facility in East Kilbride, Scotland where the world's first single-chip SmartCard MCU then went into full production.

Since then the advent of HCMOS technology and the ability to integrate electrically erasable memory arrays on the same chip have resulted in the extension of the family of SmartCard microcomputer chips based on the M6805 CPU. This family offers various combinations and sizes of RAM, ROM, EPROM and EEPROM areas. Other families of high-performance, high-functionality MCUs which have been developed in parallel with the SmartCard family, notably the 68HC05B-, 68HC05C- and 68HC11- families, are ideal for controlling SmartCard readers, keyboards, communications channels and interface devices.

Now, in 1990, East Kilbride is Motorola's world-wide centre of excellence for SmartCard product and manufacturing technology and, as a result of its ongoing commitment to this emerging market, has so far shipped more than 20 000 000 SmartCard devices.

DESIGN SECURITY

The silicon designer's task is never an easy one, but it is particularly difficult when the end product is intended for SmartCard applications. There is a continual trade-off between what the market demands and what it is willing to pay for in terms of functionality. One of the main reasons why general purpose single-chip MCUs are not a cost effective solution for SmartCard applications, forgetting for the moment the lack of in-built security, is that a considerable area of silicon is wasted supporting functions which are not really
necessary for this type of application. All they serve to do is to increase the die size which increases the cost of production, and hence the price of the end product. The relative quantities of different types of on-chip memory is another key issue. The SmartCard market has a voracious appetite for non-volatile memory. However an EPROM cell uses roughly twice the silicon area of a ROM cell, and an EEPROM cell is about twice the size of an EPROM cell. In other words, if the target die size can accommodate 8 K bytes of ROM, the customer can have approximately 4 K bytes of EPROM or 2-3 K bytes of EEPROM instead. This is very much an oversimplification as for example each type of memory has a different requirement for decoding and programming circuitry, which also add to the silicon area. Also, the combination of non-volatile memory technologies and NMOS and HCMOS semiconductor technologies makes the manufacturing and testing processes much more complex and expensive. Although technological advances have helped reduce the dimensions of non-volatile memory cells, EPROM will always be more expensive than ROM, and EEPROM will be more expensive still. RAM is the most expensive type of memory in terms of silicon area. Fortunately, very large arrays of RAM are not usually required for SmartCard applications, though this does put pressure on cryptologist-programmers to use the available RAM very efficiently.

The result of the silicon designers' efforts may well be a masterpiece of design ingenuity with a functional specification second to none, but it also has to be manufacturable, in very high volumes, at the right price for the end customer – hence it has to be testable, quickly and thoroughly. Testing of a microprocessor device is usually achieved by accessing the data, address and control buses via the pins which electrically connect the internal circuitry of the device to the outside world. This is fairly easy to achieve with standard microprocessor devices which are intended for multi-chip applications, as their data, address and control buses appear on the pins of the device, and are designed to allow easy interface to external memory and interface chips. It is considerably more difficult to achieve in most single chip microcomputers. However the design engineer usually incorporates a test mode which makes the internal bus connections externally accessible, by switching them onto the various I/O pins leading to the outside world.

Single chip MCUs intended for SmartCard applications present the silicon designer with a number of special problems. The device circuitry must be designed in such a way that, after testing, the test mode can be permanently disabled to prevent this mode from being reactivated. The device must then operate only under the control of the user application software.

Single chip MCUs for SmartCards do not need a large number of connections to interface to the outside world. (The current ISO standard, which specifies a serial half duplex system, defines six connections, plus two reserved for future use.) Although this presents the designer with the opportunity to save on valuable silicon area, it introduces the additional problem of how to provide access to the internal circuitry, when there are insufficient I/O lines available to connect the internal buses to the outside world. This problem can be overcome by adding special test connections to the device, which are not connected when the device is packaged in a SmartCard, or by utilising serial communication techniques during testing. However, extra test pads add to the total silicon area of the die, and serial communication adds to the testing time, both of which ultimately add to the cost. Motorola's designers have developed a number of techniques and features which ease the testing burden and reduce the test time of the device. These features are disabled by blowing fusible links upon completion of the test.

![Fig. 4: Outline of MCC68HC05SC11/SC21 die showing the effect of cell complexity on the respective memory areas](image1)

![Fig. 5: General Purpose Microcomputer](image2)
Fusible links can be incorporated in the circuit design to connect the I/O lines to the internal data bus and other parts of the internal circuitry. These fusible links form part of the normal interconnecting layers embedded in the device during the wafer fabrication process. After testing is complete, these fuses are blown. In non-volatile memory arrays (EPROM and EEPROM) special bits or bytes can be provided within the array which allow the CPU to detect attempts to illegally erase the memory. These bits can be programmed only during testing, before the fuses are blown. Any attempt to erase the memory array will irrevocably alter the state of these bits. In some designs, it is left to the application software to decide on an appropriate course of action if such a condition is detected. In other designs, the CPU stops processing - permanently.

The layout of the circuitry of a semiconductor device on the silicon die is usually done in a fairly logical manner. However, the SmartCard chip designer can use a number of techniques to confuse anyone who may try to analyse the design. Communication buses and control lines can be routed through different masking layers rather than being routed by the most direct path. Memory topology can be made very complex with logically adjacent bits and bytes being physically distributed over the memory space. Also, dummy structures resembling transistors can be distributed within the integrated circuitry on the die.

**WAFER FABRICATION**

Over the last 10 years or so Motorola has progressed from 5 μm NMOS technology on 75 mm wafers to 1.5 μm low-power HCMOS technology on 150 mm wafers. The significant reduction in line widths has allowed larger memory arrays, new memory technologies, and more functional circuitry to be incorporated in today's state of the art devices without increasing the die area. Unfortunately, this wonderful new technology does not come cheap. The capital investment in state-of-the-art wafer processing and test equipment is extremely high. A modern wafer fabrication facility such as the one Motorola has built in Scotland costs about £200 million to build and equip. This in itself may be viewed as an additional security feature against fraudulent manufacture!

The production process which turns raw silicon into finished wafers involves a very large number of process steps. In simple terms it is a multi-layer technology using techniques similar to photographic printing. The electronic circuit is converted into a number of layers which contain different elements of the circuit components: the interconnections; the bus lines; the bonding pads; the memory cells; and so on. Each layer is then converted into a photographic mask. (One of these masks contains the application software and customised data supplied by the end customer.) Each mask, and there can be as many as 15 in some complex device technologies, is then used in a series of exposure, print and development cycles as the electrical circuit is built up on the silicon wafer. In addition, a number of diffusion, oxidation and implant processes are used at different stages of the process to create the correct electrical and functional characteristics of the semiconductor device. The wafer fabrication process for producing secure microcomputers is essentially no different from any other type of microprocessor or microcomputer. However the inclusion of EPROM or EEPROM arrays makes the process much more complicated due to the extremely critical nature of such parameters as gate and interlevel oxide thicknesses and silicon defectivity.
MANUFACTURING SECURITY

I have just reviewed the standard wafer fabrication process. What does the semiconductor manufacturer have to do differently when the product can end up in a SmartCard application, possibly representing a considerable sum of money or controlling access to bank accounts, personal information or to high security areas?

Security must be the watchword at every stage of the process. In Motorola, for any single chip MCU, the customer’s application software (the ROM code) is converted into a set of geometric coordinates and dimensions, which are stored on a pattern generation tape. This tape is then used to produce one of the masks used in the wafer fabrication process.

For SmartCard MCUs the customer’s software is processed, and the pattern generation tape produced, via a special restricted-access account on Motorola’s internal computer system. Transfer of the tape to mask shop and of the ROM code mask and tape back to the factory is done by special courier. The mask is then safely stored within the wafer fabrication area which is itself a highly restricted area.

During the wafer fabrication process the wafers are strictly controlled on a batch basis up to the start of the metallisation process. From then on, each wafer is tracked and traceable individually, as the devices are essentially functionally complete after metallisation. As well as having various access controls to the manufacturing areas every wafer entering the metallisation stage is fully accounted for (even to the number of good dice on each wafer, once that is determined) from then on until it is either shipped to the customer or destroyed in a secure manner.

TESTING OF NON-SMARTCARD DEVICES

Every die is tested for total functionality and performance to the device specification, including verification of the customer’s application software in ROM. During the probe test any defective die can be physically marked or located on a wafer map to eliminate any further work on such non-functional devices. The wafers are then fixed to an adhesive plastic film before being sawn to separate the individual dice. If the devices are being supplied to the customer in packaged form, the wafers are then sent for assembly where the dice are assembled into the appropriate packages. They are then final tested and shipped to the customer via standard commercial carriers.

TESTING OF SMARTCARD DEVICES

In the factory in East Kilbride the SmartCard probe test area and all its test equipment is completely dedicated to SmartCard products and access is restricted to authorised personnel only. Each device is functionally and electrically tested as with non SmartCard product. The non-volatile memory is then erased and retested to ensure complete erasure of every cell. At this point traceability information (such as batch number, manufacturing location, test date etc.) can be written into each device together with any customer specific data, in accordance with the customer’s instructions or algorithms; some of this data may be unique to each individual die. This part of the procedure is carried out via a dedicated computer system which has no communication links with the world outside the SmartCard probe area. Finally the security fuses on each die are blown.

After being sawn, the dice are ready to be shipped to the customer as complete wafers on their adhesive plastic film backing. The wafers are transported from the factory in East Kilbride to the airport bonded warehouse – and from the destination airport to the customer’s premises – by armoured car. These wafers include not only good dice which have passed the test programme, but also any dice which have been identified and marked as defective. This allows the customer to account for each individual die, good or bad, on every wafer.

IN CONCLUSION

I hope that this Engineering Bulletin has given you some insight into the design and manufacture of single chip microcomputers in general. I have tried to illustrate, through the experience gained by Motorola over the past twelve years, how the additional problems set by the need for total security of design and manufacture of single chip microcomputers for the
SmartCard market have been tackled and solved. The level of expertise and overall capability required by the semiconductor manufacturer to enable him to supply the right products to the SmartCard market cannot be achieved overnight. It is not sufficient to be a world leader in microprocessor technology, with a world-beating product portfolio. The semiconductor supplier has to provide a product portfolio which includes devices designed with intrinsic security features specifically for embedding in SmartCards (or indeed in any other "security package"), and other devices with the on-chip features required for SmartCard reader and peripheral applications. Almost any general purpose MCU can be used in a SmartCard application. Some of the penalties of doing so, such as increased die size and increased cost due to unnecessary on-chip hardware and inefficient bonding pad layout, are immediately apparent; others will become obvious only when the security of such an application is put to the test, and fails to meet the necessary standards. The constraints imposed by the need for high levels of security at every stage of design and manufacture are considerable. Motorola has more than a decade of experience in volume SmartCard MCU production, and has the product portfolio required to meet all the needs of the SmartCard market in the world today. We are not standing still; we already have devices at advanced stages of planning and design to meet the emerging and future needs of this diverse market; we intend to remain ahead of the field.

GLOSSARY

BYTE (KBYTE) 8-bits of binary data (1024 bytes)
CPU Central Processor Unit – the "number cruncher" in an MCU
DIE, pl. DICE individual microcomputer (or other) device on silicon
EEPROM NVM – may be erased by applying special voltage
EPROM NVM – may be erased by exposure to UV light
HCMOS High-density low-power MOS technology
I/O Input/Output communication lines
ISO (7816) International Standards Organisation (standard concerned with specifications for IC cards)
MASK Medium used to convert customers' application software (ROM Code) to a pattern on silicon
MCU (Single-chip) microcomputer unit
MICROCOMPUTER A silicon chip containing a microprocessor – plus memory and other peripheral devices
NVM Non-Volatile Memory (for "permanent" storage)
RAM Random Access Memory – for temporary storage
ROM Read Only Memory – contents fixed during manufacture of the silicon and unalterable "ROM Code" is customer supplied application programme
"SILICON GLEN" An area stretching across the central belt of Scotland containing the factories of many international electronics companies
SMARTCARD ISO credit card sized package containing a microcomputer
TEST MODE Special operating mode for an MCU to allow comprehensive testing by the manufacturer prior to shipping to customer
USER MODE Normal operating mode for MCUs (cf TEST MODE)
WAFER Slice of silicon which, after processing, contains typically hundreds of individual dice; 100-150 mm in diameter
SCAM modules for Smart Cards

Motorola is launching its SCAM range of assembly modules by introducing its Smart Card product family packaged in modules suitable for insertion into ISO standard plastic cards.

8-contact and 6-contact modules will be available. These modules can be inserted in IS7810 plastic cards, in either the ISO standard location or the transitional location as defined in IS7816/2.

Both modules conform completely to the contact dimensions, locations and electrical connections defined in IS7816/2 (Dimensions and locations of the contacts).

Motorola is the leading supplier of secure microcomputer devices for smart card applications, with over 20 million units supplied to date from our manufacturing facilities in Scotland and the U.S.

<table>
<thead>
<tr>
<th>MEMORY (BYTES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM ROM EPROM EEPROM</td>
</tr>
<tr>
<td>6805SC01 36 1600 1024 -</td>
</tr>
<tr>
<td>6805SC03 52 2048 2048 -</td>
</tr>
<tr>
<td>68HC05SC11 128 6144 8192 -</td>
</tr>
<tr>
<td>68HC05SC21 128 6144 - 3008</td>
</tr>
</tbody>
</table>

Packaged in these modules, Motorola's secure microcomputer devices conform completely to the electrical characteristics defined in IS7816/3 (Electronic signals and transmission protocols).

During the assembly process, all pins are shorted together to minimise the risk of damage due to static electrical discharge.
"MEMORIES ARE MADE OF THIS..."
... a look at memory considerations for Smart Card applications

By Mike Paterson
Motorola Ltd.
East Kilbride

OVERVIEW

This paper discusses some of the issues concerning memory size and type and how they affect the specification of Secure Microcomputers for today's Smart Card marketplace. The principal premise is that: in the kind of mass market, multi-million unit per annum, application which is the main target of the Smart Card systems in use and under development at present, the cost is crucial to the success of a particular application.

From the silicon manufacturer's point of view cost is directly proportional to the area of silicon required (as a good first approximation at least). This paper therefore discusses the features and advantages of the different types of semiconductor memory, with particular attention to their cost in terms of silicon area, as they affect the typical uses required for Smart Card applications. After discussing some of the possible trade-offs in the specification for a secure microcomputer there is a brief look at likely future developments in technology available for these devices. Europe leads the world in the design and implementation of Smart Card based systems, from bank payment cards to access control to medical records. Motorola echoes this trend through its unparalleled expertise gained from 13 years involvement in the specification, design and manufacture of secure microcomputer devices in Europe.

This paper presents an overview of the issues concerned with memory provision and utilisation for secure microcomputers used in Smart Card applications. It does not attempt an in-depth treatment of the technical nor the applications specific issues involved.

Figure 1: The evolution of Motorola's Secure Microcomputer family
INTRODUCTION

The drive towards more and more data storage, and hence bigger and bigger memory sizes, seems to be an essential part of all modern-day electronic systems. Only a few years ago desktop “personal computers” were confidently marketed, and thankfully purchased, with only 8 kilobytes (kb) of memory. Even today the ubiquitous IBM PC has a direct memory addressing capability of only 640 kb, whilst other desktop machines are now designed to have anything from 1 Megabyte (1 Mb = 1000 kb) to several Gigabytes (1 Gb = 1000 Mb) of storage. This increase in the amount of readily accessible memory has been made possible by enormous advances in system architectures and in the way data is stored. The semiconductor memory has come a long way from the invention of the transistor fewer than 40 years ago. Today single pieces of silicon no larger than a fingernail can store more than four million bits of data, and at a cost per bit so small as to be undreamt of when the transistor was born.

The earliest “computers” used mechanical storage means (gears and cogs); this swiftly advanced to the use of the vacuum tube in the first electronic computers. The next step was the ferrite core memory – until recently still a firm favourite with military systems designers the world over. However the development of the transistor saw the realisation of physically small memory cells each capable of storing a single bit of information, a ‘1’ or a ‘0’. As the integration level (the number of transistors on a single piece of silicon) increased so then did the physical size of a given memory array fall. The decreasing physical size of memory has led inevitably in the market driven economy to a real fall in its price. Electronic goods are among the very few categories of consumer goods that have fallen in price in real terms over the last two decades and semiconductor memories have fallen (in price-per-bit terms) faster than any other commodity product.

The very success of memory design improvements and the ability to make the individual constituents of a memory cell ever smaller has in part fuelled the rush towards greater memory capacity. As more and more memory becomes available on a personal computer for example, so more and more sophisticated applications software becomes possible, which generates the need for large amounts of storage for the output of these very complex programs. The colour VDU is perhaps a good example, requiring as it does so much more memory than a simple black and white one: how often is ‘colour’ specified when black-and-white would be perfectly adequate, simply because it is readily available and has a perceived added value.

So memory is readily available, and cheap to buy, but does this necessarily mean that “more is better” is the right strategy to adopt? The danger is that the ready availability and low cost of memory may tend to obscure the benefits of making the optimum use of it. After all, no matter how small and cheap a memory cell becomes one cell will always be cheaper than two! The problem is that “more memory” is always the easy option when specifying a system and, as with colour VDU’s, you then pay for it – whether you really need it or not.

These arguments are just as valid in the context of Smart Cards as they are in the example of the desktop computers quoted above. In addition, there is a further restriction when the silicon is destined for a credit card application: size. Size can become an issue not only of cost but of reliability, as plastic bends and silicon does not! Before we look in some detail at the particular needs of the Smart Card marketplace (and by Smart Card I mean by extension all application areas where a Secure Microcomputer is desired) we first need to understand a little about the different types of memory that are available.

TYPES OF MEMORY

There are basically two categories of semiconductor memory: volatile and non-volatile. In this context the terms volatile and non-volatile simply differentiate between a memory cell’s ability to retain data in the absence of electrical power (non-volatile), or not (volatile). However both of these categories break down into sub-categories, each with different features and disadvantages.

We will look at the volatile memory first, as it is the simpler of the two, there being only two sub-categories. This type of memory is usually known as RAM (Random Access Memory) and can be divided into two main sorts, Static (SRAM) and Dynamic (DRAM). Both types of RAM are very versatile: they can be written to and read from with no special preparations; the writing/reading is very rapid; there is no practical limit to the number of read-write cycles that may be performed. These features make this type of memory useful for storage of any kind of data, from fixed program data to rapidly changing data (such as time of day for a clock function). However there are disadvantages. RAM loses its information when the power is turned off, thus it is not useful for program and fixed data. RAM cells are also comparatively large (requiring as they do up to six transistors to store just one bit of information) and consume significant amounts of power. The former means that a given amount of RAM takes up much more silicon area than the same amount of any other type of memory. The
latter varies from being a nuisance in some circumstances to a very real drawback when considering, for example, battery powered equipment.

The two types of RAM vary in the way that they store data. Dynamic RAM is smaller in area than Static RAM but needs to be continually reminded of whether it is storing a ‘1’ or a ‘0’ by means of a regular refresh (or clock) signal. Since Static RAM does not need this signal to be provided there is less system overhead, but a larger area of silicon is necessary because of the increased cell complexity. In an application therefore the system designer needs to understand whether he will always have a clock signal available when he needs to maintain RAM data. If there is any doubt about this (for example battery powered equipment frequently shuts off system clocks, whenever possible, to reduce power consumption) then he needs to specify Static RAM, and pay the penalty of increased size, and therefore price. (Static RAM has itself two variants (using either four or six transistors per cell) which allow a trade-off between power consumption and cell size.)

Non-volatile memory (NVM) is a more complicated subject. There are three main types of NVM currently available in volume production quantities with other types just coming on-stream, or currently existing in low volumes. For now we will look only at these three main types, and towards the end of this paper discuss briefly the opportunities and challenges offered by the new technologies.

The basic feature of NVM is the fact that it retains its stored data even in the absence of electrical power. This makes it ideal for permanent and semi-permanent data storage, though as we shall see its versatility is extending its use towards the RAM’s domain of frequently changing data. Again there are two basic categories of NVM: alterable and non-alterable.

ROM (for Read Only Memory), or mask-ROM as it is also known, is the simplest form of NVM. Its contents are defined during manufacture of the silicon and this data is then unchangeable over the life of the device. This type of memory has the smallest cell size (only one transistor) and is ideally suited to storing the fixed application software (operating system, program, fixed data tables etc.). However, there its usefulness ends. Since it cannot be changed after manufacture it follows that the only actions that it can perform and the only situations that the program stored in it can respond to are those thought of in advance and allowed for in the coding of the ROM contents. Because it is small and can be fully tested during silicon manufacture it is the best kind of memory to use if its features are sufficient. Even where an application demands the ability to change fundamentally during the life of the Smart Card there will be functions (such as input/output routines) which can be fixed, and hence coded in the ROM, leaving the more expensive alterable-NVM for data subject to change. There is one further advantage of ROM in the Smart Card context; because the contents are defined during manufacture, and cannot thereafter be altered, a level of security is

![Figure 2: Some typical uses for the different types of memory in a Smart Card application](image-url)
inherent in this memory. For example if an application program runs in ROM, and if this program is written such that it never allows the device to output a “secret code” stored on the card then there is no way to change this program to get the device to output the restricted data.

EPROM (Eraseable Programmable Read Only Memory) provides the next level of versatility over that of ROM. This behaves exactly like ROM when being read. However in certain circumstances the data stored in the EPROM cells may be changed. EPROM cells read as ‘1’ or ‘0’ dependent on the level of stored charge within each cell. Because of the construction of the EPROM cells it is possible, by using a comparatively high voltage, to change the state of the cell from, say, ‘0’ to ‘1’. This is a one way process. In order to return the cell to its original, or virgin state it is necessary to erase the entire memory array by exposing it to ultra-violet (UV) light. The UV light causes a shift in the threshold voltages of the transistors in the EPROM array so as to allow the stored charge to leak away and the cells to return to their original (zero) state. Obviously by its nature an EPROM array will all be erased at once, it is not possible to selectively erase particular data. Again in the case of Smart Cards this limitation can be used to advantage. The (fixed) ROM code can be made to check that some particular part of the EPROM array is programmed (i.e. is non-virgin). On failing to detect this situation the applications program knows that the EPROM array has been erased at some point and hence the integrity of any data now in the EPROM is suspect. Further operation can then be suspended, or any other predetermined action taken. Clearly also in a Smart Card it is normally not possible to erase EPROM anyway as the die is permanently covered by opaque plastic. This means that for these applications EPROM is a ‘write-once’ or ‘one-time-programmable’ type of memory.

To get round the final restriction of non-selective erasure we have to move towards the most complex, largest, and most versatile type of NVM, the EEPROM array. Electrically Eraseable Programmable Read Only Memory can behave exactly like EPROM but in addition offers the advantage of being able to selectively erase, and re-write specific bytes of data. Again the EEPROM cell, like that of the EPROM, uses stored charge to differentiate between a ‘1’ and a ‘0’. Again like the EPROM, by use of a higher than normal voltage it is possible to change the state of the cell from, say, ‘0’ to ‘1’. However this time the process is reversible and hence the cell state can be returned to ‘0’ in this example by simply applying the correct voltages to the cell. A further advantage of EEPROM over EPROM is the fact that because the programming/erasure currents involved are so small it is possible to generate the high voltage required for this on the silicon chip itself by using a ‘charge pump’, thereby removing the need for a separate high voltage power supply external to the chip. This is a significant design simplification and provides significant cost reduction in the Smart Card arena, where the number of external contacts is thus reduced from six to five. Additionally the Smart Card reader designer does not have to provide expensive, high tolerance programming circuitry like that necessary to support EPROM based cards.

However all this versatility has a price. As stated the EEPROM cell is more complex than any other type of non-volatile memory cell. It therefore takes up more space and hence costs more. Because EEPROM can be written to very easily it puts an additional burden on the system software designer in order that the integrity of the data can be guaranteed. For example: to return to our ‘application running in ROM’ quoted above. If this same application software is now run in EEPROM then it is possible to change the actions of the program (by simply rewriting some of the bytes containing that program) to provide an option that was previously non-existent. The software designer therefore has to ensure that his software guards against any possibility of an unauthorised user being able to gain control of the application program in order to make changes to its operation. Additionally further precautions, both in hardware and software, must be taken to minimise the chance of spurious data being written into the EEPROM, due to a power failure for example.

Because the EEPROM memory is so much more versatile it is consequently more open to abuse. This throws more dependence back on the inherent design of the hardware itself, as well as on the software. Security implemented as part of the hardware design of the device protects against a variety of ways that the memory contents could be compromised – for example by ensuring that there are checks after the device is reset to determine the contents of particular areas of EEPROM.

Alterable-NVM such as EPROM and EEPROM, by its physical nature, has a finite life in terms of data storage and therefore its integrity can be regarded as lower than that for the unchanging ROM cells. The “life” of EPROM or EEPROM is usually given in terms of data retention and write-erase endurance. Data retention is simply a measure of how long any particular information is guaranteed to be retained in memory after being written there with a given set of conditions (voltage, programming time, temperature etc.). Write-
erase endurance is quoted as the number of times that any individual cell may be written to and erased and still be guaranteed to correctly store data. Typical values are: 10 years for data retention; 10 000 write-erase cycles for EEPROM endurance. EPROM is a more mature technology than EEPROM, and needs comparatively high currents to perform the programming (electrical erase being impossible). Therefore the data integrity of EPROM is generally regarded as being better than that of EEPROM, there being a lower probability of incorrect data being stored (for good data being erased) through the action of "electrical interference" (power surge/failure, etc.). There are of course ways to improve statistically the effective reliability of data stored in EPROM or EEPROM, for example by using intelligent programming algorithms, or by using error detection and correction techniques. Each of these methods can be implemented so as to achieve the most demanding of data integrity requirements. Hence the recommendation should be that: use the flexibility of EEPROM if it is needed; but if "write-once" EPROM is sufficient for the nature of the data being stored then use it in preference; and if the data is fixed then use ROM.

We can summarise the memory types as follows:

**RAM**
- The fastest and most versatile type of memory; volatile

**EEPROM**
- Slower than RAM, but retains data in the absence of power

**EPROM**
- For Smart Cards this is really a write-once kind of memory

**ROM**
- Data defined during silicon manufacture and then unchangeable

### MEMORY USE IN SMART CARD APPLICATIONS

Before we can look even briefly at the type of uses that memory is put to in a Smart Card application we first need to define "Smart Card". In this context I shall take a fairly narrow and literal definition. Thus the Smart Card is a device in ISO 7816 credit-card form containing a single-chip microcomputer. There are many other possible implementations which could give the same functionality, for example multiple-chip solutions, or key-fob shaped packages, etc.; the adoption of the above definition is merely to make the discussion simpler.

Whilst the possible applications for Smart Cards are limited only by the imagination of the systems designers in their ability to demonstrate added value, a number of common basic features are required. A Smart Card is by definition and intention an identification and authentication medium. No matter what actual use the card is being put to its fundamental purpose is to authenticate the card and its owner and to establish their level of entitlement; secondary functions such as payment, personal details, or data transfer are then used dependent on the nature of the application. This leads us to the basic common requirements of the secure MCU in the card: it should have the ability to store the identity of itself and its owner in such a way as to prevent the unauthorised use of this information. The method commonly, if not universally adopted at present for positive identification is through the use of a Personal Identification Number (PIN) together with a scrambling (or coding) function of some kind. Hence the individual card needs to have the ability to manipulate numbers according to some encryption system and to store in a secure manner unique data pertaining to that card and its owner.

In order to perform any data manipulation "scratchpad" memory is required. This is memory that may be written to and rewritten countless times during the progress of a single calculation or transaction. Since there is also a need for this memory to be accessed as fast as possible, to minimise calculation times, then the only choice is RAM. However, as I described earlier RAM is the most expensive kind of memory, due to its large physical size. This means that the system designer needs to pay a lot of attention to minimising the use of the "scratchpad" at any one time, to keep the percentage of the silicon area dedicated to temporary results (often of no consequence in themselves) as small as possible. The biggest use of RAM by far in a Smart Card device is for storing the intermediate results generated by the encryption or decryption algorithms and it is here that there is the biggest potential payback in terms of cost savings from efforts to optimise the use of RAM.
Alterable-NVM has the next biggest cell size after RAM, whether we are talking about EPROM or EEPROM. A fundamental requirement of a Smart Card, for whatever application, is that it can be "personalised" or customised for each individual user. This requires NVM since the individual data will have to be written immediately prior to the card being issued to a particular individual; and it will of course have to retain that data when the card is not in use and not powered up. The arguments between EPROM and EEPROM are complex ones. The significant functional difference between these two types of memory is that (at least in credit-card packaging) EPROM is a "write-once" medium, whereas EEPROM can be written and erased many times.

If write-once versus repeated write-erase was the only issue then obviously EEPROM would win every time. The flexibility of EEPROM is unchallenged. Where the exact future usage of the memory is not known and cannot be predicted then it offers the best choice. However if the use of the memory is known and can in fact be used in a write-once mode then EPROM has a number of advantages, not the least of which is cost. Bit for bit EPROM is less than half the size of EEPROM, offers a higher level of intrinsic security (more resistant to unauthorised alteration since erasure is not possible in a Smart Card application), and is a more mature technology with the associated benefits of reliability and predictability. By making use of individual EPROM bits the number of transactions that can be sequentially recorded before the memory fills up can be quite large, however then the card has to be thrown away whereas the EEPROM card can simply have obsolescent data erased as necessary, and the memory re-used.

There are two main uses for the NVM: frequently updated data (such as information on every transaction); and only occasionally updated data (such as a new PIN code, credit limit, personal details, etc.). For the latter category the availability of more than twice as much EPROM memory in the same area may well make it equally or even more attractive than EEPROM. In the former situation however the ability to continually "refresh" the EEPROM card’s capacity is an advantage, avoiding as it does the "throw away card" concept. However it should also be remembered that the lifetime of a credit card is finite and controlled by such considerations as plastic wear, customer expectations, and even security. A mixture of EPROM and EEPROM functionality would appear to be desirable, and this development is discussed later.

Figure 4: A comparison in schematic form of the circuitry involved in implementing ROM, EPROM, EEPROM, DRAM and SRAM cells in MOS technology suitable for microcomputers

698
The final memory type is the fixed ROM which can contain the applications program and general routines for input/output, PIN change etc. There is at least the temptation (and not only in Smart Card applications) for system designers to use the versatility of EPROM & EEPROM almost as a crutch, enabling last minute changes to software to be made "painless". Whilst this does mean that the need for fully functional final software is postponed as long as possible, this delay does carry a price as we have seen, in terms of silicon area. It makes economic sense to ensure that as much of the software as possible is fully defined before the manufacturing of the die starts. In this way the fixed data can be most efficiently stored on the silicon while enabling the available amounts of alterable-NVM, be they EPROM or EEPROM, to be reserved for functions that really require its additional features. In a typical Smart Card type of application we might reasonably expect to see the following apportionment of the software in non-volatile memory: ROM – fixed or "core" software (e.g. I/O routines, read/write sub-routines, basic encryption algorithm, planned use software (e.g. bank debit card); EPROM/EEPROM –

sections of the code which it may be desired to update on a 'regular' basis (e.g. algorithm seeds, precise flow of the algorithm, tracking credit balances & expenditure profile) or which are unique to each individual card (e.g. serial number, personalisation data). EPROM or EEPROM can also be used to allow new functions to be loaded into the card after it has been issued to individual users (e.g. medical data, use as a credit card, telephone prepayment card, access control). By ensuring that the basic routines and functions are stored in ROM it is then possible to change the exact way in which they operate by simply writing different linking software to be stored in the E/EEPROM. For example it would be possible to operate the data transfer to and from the card at different rates based on a single number stored in E/EEPROM used in conjunction with the basic routine in ROM.

Figures 4 and 5 show in some detail the circuitry involved in typical memory cells as they are constructed on microcomputer devices together with a comparison of the actual layout areas on silicon for some of these memory types.
VERSATILITY versus COST:
THE MEMORY TRADE-OFF

If we take the size of a given ROM array to be 1 then the relative sizes for the other types of memory array are approximately: EPROM – 3; EEPROM – 7; dynamic RAM – 15; static RAM – 30. The reasons for this increasing size are simply the increasing complexity of the cells needed to provide the features of each type of memory, plus all the associated driving and decoding logic.

![Figure 6: Relative memory cell sizes](image)

THE FUTURE?

Clearly the Smart Card market, and indeed the market for secure microcomputers as a whole is still in its infancy. There are a number of very large projects already well underway—in France, Germany, Norway, Switzerland and the U.K., for example, but there are countless more at the stage of advanced planning or preliminary trials. The future for the actual microcomputer chips themselves will be largely determined by the way this marketplace develops. Criteria such as die size, memory size, level of designed-in security, hardware features and so on will all be dependent on which of the many proposed applications actually stay the course and become volume users of this technology.

Without the ability to predict the future and say which will be the most significant uses of Smart Card technology in the 1990’s and beyond all we can do is look at current trends in both technology and applications and assess their likely development in the next few years.

It is probable that we will see a reduction in the (apparent) homogeneity of the market; there will be requirements for very large memory array devices, which will be comparatively expensive (multi-function financial cards for example); there will also be a need for very cheap “stripped down” devices, aimed at providing a single, well-defined service very economically (perhaps a tracking token used for manufacturing control, or a simple access control card). Whilst it would appear at present that the latter category has the greater potential for generating very high volume business it is already clear that one of the major strengths of the “Smart Card” is its ability to be a truly multi-purpose card—where further new functions (and even distinct applications) can be added even after the cards have been issued to the individual end-users.

However, to return to the topic of this paper, the short-term silicon technology developments which we can expect to see in support of the above market forces are basically to do with putting more and more functionality in a given area of silicon. Thus memory cell sizes will continue to shrink, and a significant reduction in this size could be in the use of “flash memory technology”.

Flash-EEPROM is a modification of the existing EEPROM technology which reduces the effective individual memory cell size by abandoning the ability to individually erase specific bytes of data. In order to reduce the connection overheads for each cell the flash-EEPROM can normally only be erased in bulk (i.e. the entire array) or as a relatively small number of fairly large data blocks within the array. By making this seemingly simple change it is now possible to get almost as many bytes of flash-EEPROM memory as EEPROM memory in a given area of silicon. It must be borne in mind though, that flash-EEPROM is a compromise technology: it will not be as small (hence not as cheap) as EEPROM; it will not have the high write-erase cycling endurance of true EEPROM (perhaps only 10 – 100 changes of data, rather than the typical guaranteed figure of 10 000 for the EEPROM); nor will it have the selectivity to be able to erase/rewrite single bytes of data, with all the speed advantages that implies; finally it will require an external high voltage source to perform the erase as the bulk nature of this erase precludes the use of an internal charge pump, since the required current is too high. As far as the Smart Card type of applications are concerned then flash-EEPROM will give the system designer some of the benefits of EEPROM eraseability, with a cell size (and therefore cost) approaching that of EEPROM.
Just becoming available on the market now is a new generation of microcomputer devices which combine both EPROM and true EEPROM on the same die. This mixing of manufacturing technologies has not been trivial but it offers unparalleled versatility to the system designer: he has the write-once kind of changeable memory to add, for example, a new application or a correction to an unforeseen limitation of the software; he also has the full write-erase cycling capability of the EEPROM to cope with frequently changing data, such as credit balances, entry logging etc. Because both types of memory are on the one piece of silicon it is possible to specify the minimum of the more complex and expensive EEPROM as it will only be used for data that changes many times and will not have to cope with the change-once data also, as the current EEPROM only devices have to.

Other developments for the near future which will affect the memory requirements of the secure microcomputer are likely to be in the field of algorithm handling. As we move to more and more sophisticated algorithms so the amount of processing power required increases and also the amount of working storage (RAM) necessary. It is presently a challenge to the cryptographers to cram the interim results of their encryption or decryption into the available RAM. Since RAM is expensive on silicon the pressure is unlikely to ease. Further advances in RAM technology and further shrinking of the cell size will increase the amount of available RAM; however by then the algorithms will require more working storage to cope with their ever increasing sophistication.

Finally we can expect to see the “contactless Smart Card” becoming more common as the complexity and cost problems of this technology are overcome to enable it to compete in the volume marketplace. Presently the potential disadvantage of contactless solutions is that the card has to contain at least two chips, with the associated issues of manufacturability, cost, reliability and security. One development that is firmly in the silicon manufacturer’s court is to devise the means to combine the standard HCMOS process technology used for microcomputers with the RF technology needed for the transceiver function of the contactless card. Once this can be achieved then a single-chip contactless Smart Card will be a reality for the high volume low cost general marketplace, and at a stroke many of the interfacing and mechanical contact issues will be circumvented. This is not likely to happen in the immediate future, but if the demand is real then it will happen.

So in summary we can see that there is enormous scope for development of the secure microcomputer. As more flexibility becomes readily available this will of course make more and more potential Smart Card applications viable. The credit card format is a very acceptable one to the end-user, and it may well be that this is the shape of things to come. However one of the reasons that the Smart Card is so acceptable to users is because of the enormous numbers of “credit cards” for a wide range of services already being carried around in people’s wallets. When the only kind of “credit cards” are those containing a secure microcomputer perhaps we will see a new, more compact and more durable form of personal identification medium.
<table>
<thead>
<tr>
<th>FEATURES</th>
<th>ROM</th>
<th>EPROM</th>
<th>EEPROM</th>
<th>RAM</th>
<th>FLASH EEPROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative array size on silicon</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>20–30</td>
<td>3–6</td>
</tr>
<tr>
<td>Number of write-erase cycles</td>
<td>1</td>
<td>10–20</td>
<td>&gt;10 000</td>
<td>∞</td>
<td>100</td>
</tr>
<tr>
<td>Data retention time</td>
<td>∞</td>
<td>10 yrs</td>
<td>10 yrs</td>
<td>as long as power on</td>
<td>10 yrs</td>
</tr>
<tr>
<td>Program voltage</td>
<td>N/A</td>
<td>external</td>
<td>internal</td>
<td>(internal)</td>
<td>external</td>
</tr>
<tr>
<td>Erase voltage</td>
<td>N/A</td>
<td>UV</td>
<td>internal</td>
<td>(internal)</td>
<td>external</td>
</tr>
<tr>
<td>Write time</td>
<td>N/A</td>
<td>5 ms</td>
<td>10 ms</td>
<td>bus speed</td>
<td>2–10 ms</td>
</tr>
<tr>
<td>Erase time</td>
<td>N/A</td>
<td>minutes</td>
<td>10 ms</td>
<td>bus speed</td>
<td>seconds</td>
</tr>
</tbody>
</table>

Figure 7: A comparison of the principal features of each memory type
<table>
<thead>
<tr>
<th>Glossary Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>a single item of information; either “0” or “1”</td>
</tr>
<tr>
<td>BYTE</td>
<td>eight bits of data</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic RAM – requires a refresh, or clock, signal</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Eraseable Programmable Read Only Memory</td>
</tr>
<tr>
<td>EPROM</td>
<td>Eraseable Programmable Read Only Memory</td>
</tr>
<tr>
<td>kb, Mb, Gb</td>
<td>kilo- (1 024 bytes), Mega- (1 048 576), Giga-byte (1 073 741 824)</td>
</tr>
<tr>
<td>FLASH</td>
<td>“Flash EEPROM” – a limited form of EEPROM</td>
</tr>
<tr>
<td>HCMOS</td>
<td>High density Complementary Metal Oxide Semiconductor; a manufacturing process for MCU’s</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organisation</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro Computer Unit</td>
</tr>
<tr>
<td>NVM</td>
<td>Non-Volatile (“permanent”) Memory</td>
</tr>
<tr>
<td>PIN</td>
<td>Personal Identification Number</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory: temporary storage</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
</tbody>
</table>
INTRODUCTION

What is a “Smart Card”? There are many definitions for this term, probably one for every article ever to appear in the popular press! However, before I define just what I consider to be a “Smart Card”, I would like briefly to explain my choice of title and summarise what I hope to cover in this Engineering Bulletin.

For a number of years it has been fashionable to consider Smart Cards as a “solution looking for a problem”, or as an “invention searching for a market”. These are now becoming less fashionable expressions as the market for Smart Cards grows and diversifies. However, it is this historical basis and the very diversity of potential applications that are now beginning to come to fruition, which are in some ways hampering the growth of the marketplace.

This Engineering Bulletin sets out to explain, from a silicon manufacturer’s perspective, the background to the current market and the variety of products that are on offer. It defines the “Smart Card” and looks at the ways to determine what a particular application needs, in terms of features, if a Smart Card solution is going to succeed for it. By its nature this Bulletin provides only an overview of the many, and complex, arguments that accompany any decision to introduce a Smart Card application, but it is hoped that this will give some basic guidelines to help the prospective end-user to ask his supplier the right kind of questions about the product types available. By choosing the optimum product for the particular market the likelihood of commercial success in that market is enhanced. So, by asking the right questions you end up with the right product. Remembering that (like Poker) you do not see your competitor’s hand until it is actually played, it is clearly advantageous to be holding all the Aces before you place your bets on success.

Motorola East Kilbride has been the company’s worldwide “Centre of Excellence” for Smart Card products for more than ten years. Together with their Geneva Design Centre they have developed and supported a range of secure single chip microcontrollers for Smart Card and other security applications. To date more than 20 million secure MCUs have been sold and Motorola now has customers throughout Europe, Japan and the Americas using its Secure MCU product range. The author has worked in Product Engineering for several years on supporting this range of device types.
WHAT IS A SMART CARD?

The simplest definition might be that it is capable of “thinking” for itself – in other words, it has built-in computational ability within the credit card itself which can modify, and even create, data in response to external stimuli. This, then, distinguishes the microcontroller (MCU) based card from all other types. This definition, and its isolation of the MCU based card, is the one I am going to use throughout. All other cards are therefore “dumb” as opposed to “smart”. However, it is only fair to say that there are varying degrees of “smartness” and “dumbness”. The optical cards for example, with their vast memory capacity, are capable of deploying great amounts of data in order to enhance their operation. However their operational responses are rigidly defined and they cannot create any data: that requires an external “computer” to be connected on-line.

TYPES OF “CREDIT CARD”

The magnetic stripe card has been around for many years now and it is typically against this bench-mark that all new card technologies are measured. Before we look at how the Smart Card compares with the magnetic stripe card, it may be useful to list what types of card are available and look briefly at their advantages and limitations.

<table>
<thead>
<tr>
<th>Type of Card</th>
<th>Cost</th>
<th>Capacity</th>
<th>Versatility</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>- embossed plastic</td>
<td>low</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>- with hologram</td>
<td>low</td>
<td>nil</td>
<td>nil</td>
<td>low</td>
</tr>
<tr>
<td>- with magnetic stripe</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>- with non-volatile memory</td>
<td>medium</td>
<td>medium</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>- with battery-backed RAM</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
<td>low</td>
</tr>
<tr>
<td>- with MCU</td>
<td>high</td>
<td>medium</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>- with multi-chip solution</td>
<td>high</td>
<td>medium</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>- with optical storage</td>
<td>medium</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>- with mechanical storage</td>
<td>low</td>
<td>low</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

Table 1: A comparison of Card types and their features

The types of card available are:

1. Embossed plastic (with or without hologram or other visual security feature)
2. Card with magnetic stripe containing personalised/security data
3. Card with optical storage medium
4. Card with physical/mechanical storage medium
5. Card with silicon chip
   a) memory only
      i) EPROM (write once)
      ii) EEPROM (multiple write)
      iii) Battery-backed RAM
   b) MCU (with various memory types)
   c) multi-chip (“contactless cards” etc.)

Of course not all of the above categories are exclusive, indeed many cards currently are a combination of (1), (2) and (5).

As Table 1 shows, the main differentiators between the various types of card are: cost; storage capacity; versatility; and security.
**SMART CARD VERSUS MAGNETIC STRIPE**

The Smart Card IS NOT a replacement for the magnetic stripe cards that are commonly to be found in all our pockets, at least not directly. The magnetic stripe card has two overwhelming advantages: it is cheap; and it is established – there are hundreds of thousands, if not millions, of standardised magnetic stripe card readers all over the world. This second advantage is, of course, held over all possible competition, and is a temporary hurdle, which will ensure that the new standard (whatever it is) really does offer some distinct advantages! However, on the issue of card cost alone, it is quite safe to say that the MCU based Smart Card will never be as cheap as a magnetic stripe card.

Given the above facts it is essential to look for value added from a proposed Smart Card implementation. This may either be direct, such as enabling a previously impossible application, or indirect, for example in reducing losses due to fraud or misuse.

So, the Smart Card is more expensive than a magnetic stripe card. On the other hand, it surpasses the magnetic stripe card when any other feature is considered. It has many times the data storage capacity, it offers a much higher degree of transaction security and data integrity, and it is flexible – it can support more than one type of transaction, and can be reconfigured after it has been issued to the end-user. If your application needs only the features of the magnetic stripe card then it is unlikely that a Smart Card system will succeed for you. If, however, the application would benefit from the increased flexibility for example, and this can be quantified, then it is much more likely that the system will be commercially viable.

**FEATURES OF A SMART CARD**

As I have already said, the benefits of the Smart Card are only relevant if the proposed application gains something from them: it is not necessary to play an "ACE" to beat your opponent's "TWO", a "THREE" will do. Nevertheless, let us look at the strengths of the Smart Card, then later we can determine what a specific application actually needs.

I would list the principal features of Smart Cards as follows (in alphabetical order, rather than by merit!):

- Anti-fraud Capability
- Continuous Application and Transaction Validation
- Cost
- Flexibility
- Multi-purpose
- Off-line Capability / "Stand-alone"
- Positive User Authentication
- Reconfigurable In Use
- Security
- Speed
- User Friendly

As you can see, I feel that even overall cost can be considered a feature, when looked at in terms of initial costs, operating costs and cost effectiveness!

Many of the features are, of course, inter-dependent and so I will try to discuss the overall impact of all of them as well as looking at them individually.

Because of the Smart Card's ability to perform its own validation checks, and to store details of transactions internally, it is at once secure and capable of operating in a "stand-alone" mode, that is, with no need to communicate with a central computer/database for every transaction. It is this ability to store data securely, and to operate on that data and on external data within the card, that distinguishes the Smart Card from all other "data cards".

Security in the application is on many levels. There may be the normal software controls, such as passwords or PINs. There are hardware monitoring functions on the silicon itself (to protect against bulk erasure of data for example). There are also the specific hardware features of a Secure Microcontroller which effectively prevent access to data stored on the chip, except in the very limited way permitted by the application software, to ensure that data cannot be selectively modified. By making sure that all the links in the security "chain" are strong ones it is possible to rely on the integrity of the data stored in the die – and hence to perform authentication and validation tests within the card. The benefit of this feature is that the card can perform its own validation of any system it is connected to. This places requirements in terms of security and traceability considerations from the initial design of the chip itself, through silicon manufacture, to system software, and ultimately features of the application.

Given that the data storage is accepted as being secure, the card can then retain information about, for example, a person's credit balance (and balance remaining at a given time) as well as storing PIN data.
or passwords, to avoid the need for a central computer link-up. It can also store usage patterns, voluntary limitations of use, and any number of specific details—so that it becomes possible, for example, for the card itself to initiate a fraudulent use check if the usage changes suddenly. The internal computing capability may also be used to detect fraudulent terminals, by continuously exchanging information cryptographically and so validating the terminal to which the card is connected.

**ANTI-FRAUD CAPABILITY**

Because there is intelligence in the card, if it is used in a Credit Card application it can prevent overspending by its authorised users, unlike the conventional magnetic stripe card. This is simply because the credit limit and the current balance can be stored on the card, making the response time to overspending immediate and independent of human interaction.

Biometric data is perhaps the most visible of the new techniques being employed to avoid fraudulent use. Though by no means restricted to Smart Cards, the ability of the card itself to allow for changes in response (for example voice change due to a cold) make it a powerful tool. The basic advantage here for the Smart Card is that it can compare stored data with actual data in complex ways, and relate many different pieces of information. For example, there are always tolerances allowed for in biometric data. The card can monitor and update tolerance levels as it "sees" your signature change over a period of time. It can also ask for reconfirmation of identification in cases where a large amount of money, or entry into a high security area, is at stake, especially if the biometric data is on the extreme limit of acceptability.

In addition to all the security features on the silicon and in the software, it is possible for the card to store "typical usage" data. This then allows for the possibility of triggering a "fraudulent use check" if the usage suddenly changes.

**CONTINUOUS APPLICATION AND TRANSACTION VALIDATION**

The card can re-verify at any time that it is connected to a legitimate system and application by requesting security data from either the user (eg: biometric or PIN data) or the system (eg: en/decryption keys).

**COST**

Clearly the unit cost of a Smart Card is much higher than, for example, a magnetic stripe card. However, the initial cost can be offset by such factors as: the ability to reconfigure the card after issue; the capacity to store data for several distinct applications (eg: parking meter, access control, prepaid ticketing...); reduced loss of issuer income due to fraudulent use or fraudulent duplication; and the ability to function in stand-alone mode, thereby reducing system dependence on expensive, permanent and rapid access to central computing facilities for authorisation and validation etc.

**FLEXIBILITY**

The card can have several distinct applications stored on it at once, with safeguards to ensure that there is no unwanted "sharing" of data. It can be re-programmed (for example with a new credit balance, or even an entirely new application and entitlement) after issue to the end user—without recalling the card.

The exact way in which the card and the system interact is controlled by the software stored in the system and in the card. Obviously the system software can be changed in any card-based system. An advantage of a system based on a Smart Card is that the card software, and hence the form of its responses too, can be changed at any time, by simply reprogramming part of the non-volatile (i.e. the changeable) memory in the card. This means that any operating deficiencies can be corrected, or any newly conceived and valuable features can be added.

**MULTI-PURPOSE**

The number of applications installed on a card at one time is limited only by memory space. The card can respond "intelligently" to question-and-answer sessions enabling it to deal with an indefinite number of different application protocols.

**OFF-LINE CAPABILITY / "STAND-ALONE"**

The card can validate both the user and the system. It can store credit balances, area access authorisations and records of recent transactions. Standard transactions (perhaps defined as fitting the usage pattern and lying within predefined limits) may be authorised, completed and recorded, for later
communication to a central computer. "Unusual" transactions can still require the card to seek external authorisation in certain circumstances if required.

This capability both reduces cost and increases transaction speed.

**POSITIVE USER AUTHENTICATION**

Biometrics are beginning to be used, the limitations at present being principally memory size and response time. Even without this feature, however, there is the simple benefit of the user being able to define his own PIN number, including the length of it and its exact format, and being able to redefine it when and where he wants. There are, of course, varying degrees in-between these extremes of PIN and biometrics.

**RECONFIGURABLE IN USE**

If an application is compromised, or just upgraded (an extra subscription TV channel for example), then this facility can be uploaded on to the card the next time it is used with access to a host computer. This means that it is transparent to the end user, as well as being easy and cheap to accomplish – and fast.

**SECURITY**

This technology can increase security in many ways. These include designed-in chip security features (such as fusible links and illegal frequency detection circuitry); reliable tracking and safeguarding, through a controlled manufacturing process; and features written into the user software (such as a check on Reset to ensure that a particular byte of data is present). All this is possible, and moreover, because the hardware security is there it is possible to build up to very high degrees of additional software security.

**SPEED**

Waiting for a telephone link to a central computer is both expensive and slow. Local transaction processing is cheaper and faster. It is also possible to respond rapidly to a new business opportunity (such as being able to use your telephone card in another country, now that there is an agreement on national standards) by simply updating the cards already issued, rather than having to offer new ones, with all the related cost and time delay implications.

**USER FRIENDLY**

The user can: change PIN data; update personal data; customise operation; define voluntary limits; find out remaining credit/outstanding debit balances... In other words he can feel much more in control of the way the card works, which adds to its perceived convenience.

**WHAT DO YOU NEED?**

As I have already implied, all these features are only worth paying for if they pay for themselves.

As in every area of commerce we have now to look at Cost versus Features. In this particular area past experience has shown that one of the critical parameters to consider is the cost of failure. This means, for example, how much of your revenue will you lose if the security of your product is compromised; or, how much bigger is your potential market if you offer this additional feature (or conversely, how much bigger will your market be if you forego this feature and reduce the price)? The inherent security of the Smart Card solution is advantageous here as it allows you to plan your revenue with much greater confidence.

From the silicon manufacturer’s viewpoint the cost issue is quite straightforward; cost is roughly proportional to area of silicon. To minimise the cost of the chip, therefore, you have to trade off features which take up significant amounts of silicon against their expected payback in use.

The biggest single element of any microcontroller die is the memory. The memory area can typically take up to 80% of the total die area. This is the first region where you need to look for cost effectiveness. Amount of memory, and type of memory are crucial. By considering the cost of memory from the outset it is possible to structure at least some of a system’s memory requirements to make use of the most compact types of memory. For example, the more of the system software that can be fixed (and therefore stored in ROM as opposed to EPROM or EEPROM), the bigger the potential savings. Of course this does reduce flexibility from the point of view of being able to completely rewrite the system software in the card after it has been issued.
The frequency of change of the data determines the type of memory required. Small amounts of rapidly changing data will be stored in the relatively large RAM cells; EEPROM can cope with large amounts of data being updated several thousand times; the smaller EPROM cells can only cope with one change of data; and the data in the (physically smallest) ROM cells are fixed during manufacture.

Reliability and data integrity is also an issue. All changeable media, be they semiconductor, magnetic or otherwise are subject to loss of data. This again leads to the use of “permanent” (i.e. fixed, ROM) memory where possible. Once the data in ROM has been verified during manufacture, its contents are known and cannot change. For RAM, EEPROM and EPROM the functioning of the cells can, of course, be checked during manufacture but it is impossible to predict the behaviour of an individual cell under the widely varying conditions and long time periods typically to be found in a Smart Card system. It is, therefore, wise to incorporate into the system software (and perhaps the hardware too) a variety of error detection and correction techniques, exactly as is always done with magnetic recording on disk or tape. In this way the reliability and data integrity perceived by the system user is much higher.

Having considered the pros and cons of the various memory types, and also looked at the benefits of offering particular features in a specific system the next step is to optimise the card specification, and hence the silicon specification. Much of the operating code can be fixed from the outset. Where it is necessary to retain flexibility, control of how the fixed routines are linked together, and the exact parameters used, can be stored in alterable memory; this lets cards be changeable up to the point of issue, and beyond. The security of the overall application is controlled by features of the silicon and of the software. Having looked at the costs of the system being compromised it is easier to assess what degree of security is necessary. By writing the software to include a unique identity for every card it is possible to minimise the potential loss if one card is stolen; in other words, the fact that one card is compromised does not necessarily compromise the system. This effectively makes your system immune from illegal card duplication.

Communication between the card and the external system is well defined for ISO type systems. Nevertheless, it is possible to optimise the operation of your card for the desired application, so that for example, the most efficient data transfer rate and format is used for the kind and quantity of data that you are typically dealing with.

Another feature which may prove valuable in certain applications is the ability of the silicon manufacturer to identify each die uniquely during manufacture. This provides traceability information, which can have many uses: serial number; manufacturing date (plus manufacturer, test conditions, ...); a unique encryption key; etc. Because this data can only be written during manufacture, and thereafter is fixed, it enables a reliable user identification and authentication arrangement, whereby individual users can be traced back to individual cards with no possibility of error. Such data can also prove useful in analysing failure patterns, as it allows the contribution of the silicon to be evaluated very quickly, by showing for example that it is all from one manufacturer.

Finally, I return to the cost of failure. What is the “unit worth” of one of your proposed cards? If it is a card which may be used as an “electronic purse” and be preloaded with up to £1000 then the unit worth is apparent. Here it is obviously of paramount importance to ensure that the current balance cannot be tampered with, or recorded erroneously. There is considerable need to ensure that the reloading of a new balance is protected by very high security procedures. The capability of the Smart Card to perform system validation, and respond to authentication requests from the system, is crucial. If however the card is used as a “subscriber authorisation” for something such as Pay TV then the “unit worth” of a card is fairly low – a few pounds a month for the validity period of the card. Even in this circumstance it is very important that the overall integrity of the system cannot be challenged. If, for example, it is possible, by “breaking in” to one card, to compromise the whole system (by, for example, being able to issue many fraudulent cards) then the unit worth of the individual card is very high. By putting in place appropriate security and traceability features on the card it is possible to ensure that defrauding one card, if it were possible, will be limited to that one card. Hence, for security and revenue reasons, it is generally desirable to limit and minimise the unit worth of an individual card.
THE RIGHT QUESTIONS – to ask about the silicon and the system

If we assume that you have asked yourself the right questions about what your application really needs to succeed, then we can turn to the questions you should ask your card or system supplier – to make sure that the features you have decided are important to you are encompassed by the system you finally get.

SECURITY – is my revenue safe? Is my reputation safe? What features of the silicon/card/system guarantee the particular security needs of my application?

MEMORY SIZE – options and costs?

VERSATILITY – Ease of use and expandability of the system?

DELIVERY – when can I have samples? ...in volume?

QUALITY – can I depend on the product specification?

DEPENDABILITY – supplier’s track record and guarantees?

UPGRADE PATH – supplier’s history and plans?

COST – is it competitive? What am I paying for? Do I need it all?

As with buying anything, if you know what you want and why, there is a much better chance that you will get it!

THE FUTURE

From the point of view of silicon for Smart Cards there seems no reason to doubt that progress will continue. The silicon chip will continue to get smaller, cheaper, more versatile and with a wider range of features. Likely enhancements in the not-too-distant future include:

• wider supply voltage range (< 3V to > 6V?)
• lower power consumption
• faster clock speeds
• more memory
• mixed memory (EPROM & EEPROM)
• flash EEPROM
• greater system integration (e.g. RF transceiver elements on same silicon as the MOS micro-processor and memory)

There will also be a drive towards more and more complex cards, to support many applications, or to support a very sophisticated real-time data processing application, perhaps for example involving complex biometric data. This means that, in addition to the above mentioned “smaller, cheaper” trend there is a simultaneous trend to “bigger, faster, more powerful” (and therefore unfortunately, more expensive). This may give rise to a slightly different set of advances in silicon for Smart Cards:

• real-time processing capability (> Digital Signal Processors?)
• dedicated hardware encryption engines on silicon
• parallel data I/O
• much more memory

Of course there is no reason why all future developments need conform to the “credit card” format. Indeed, we are already seeing applications for “secure microcontrollers” which are built in to systems and may therefore be more conventionally packaged. However, it is fair to say that, for consumer applications, the CARD has much in its favour, whether it is with or without contacts. The package problems encountered with the “credit card” are likely to become less and less significant as power consumption continues to fall and as chip architecture continues to shrink.

IN SUMMARY

Know what YOU need – specify it, and insist on it.

As a service provider, only you understand all the needs and limitations of your business. It is easy to ask for “some of everything”, but this will not necessarily make your system succeed. If you have quantified the benefits and shown that you do indeed need a Smart Card system, then make sure that it is optimised and specified for its intended use. You can then ask the vendors what they can offer to meet your specifications – and you can ask them the right questions to determine whether what they are proposing is indeed suited to your application. Extra features mean extra costs, and so need extra revenue.

If you quantify the benefits and optimise the product specification to bring about these benefits then you have done the best with the cards dealt to you. If in addition you ask the right questions of your suppliers then you may even draw an ACE.
MC68HC705T3 Bootloader

By Peter Topping
Motorola Ltd., East Kilbride, Scotland

INTRODUCTION

Figure 1 shows the circuit required to program the EPROM of an MC68HC705T3 from an external EPROM. There is a direct correspondence between the addresses in the external EPROM and the memory map of the MC68HC705T3. The OSD characters should be in the area $0400-$0AFF, the program between $2000$ and $7EFF$ and the vectors between $7F00$ and $7FFF$. It should be noted that these addresses must be used even if the MC68HC705T3 is being used to emulate the MC68HC05T1 or MC68HC05T2. The addresses of the vectors (and the program start address for the MC68HC05T2) must thus be changed from their normal position when emulating these devices. It should also be remembered that MC68HC05T1 and MC68HC05T2 emulation should not employ any resources not present on the target devices. The MC68HC705T3 has more OSD buffers (2 rows) and characters (112), more RAM and more ROM.

The bootloader code in the MC68HC705T3 has 4 modes of operation, selected by switches S1 and S2. In addition to EPROM programming and verifying it is also possible to load and execute a program in RAM locations $0100$-$01FF$. Like the EPROM programmer, the RAM loader transfers data from the corresponding addresses in an external EPROM; there is no serial load facility.

The progress of the loader can be observed in more detail if LEDs are connected to all the pins on port D, in a manner similar to that shown in figure 1 for bits 0 and 7. If this is done, bit 7 is still used for verification and should be a different colour. During EPROM programming the high order address byte is displayed. For RAM loading the 7 lowest addresses are shown. Address 7 is not displayed in order to avoid a false impression of a successful verify.

OPERATING PROCEDURE

1. The Vdd supply should be off, S3 closed (reset) and S4 in position I (internal).
2. Insert MC68HC705T3.
3. Switch on Vdd. If this is not done before step 4, the MC68HC705T3 may be damaged.
4. If EPROM programming is required, switch on Vpp by changing S4 to position E (external); it is assumed that an external Vpp (18.5V) is present. Vpp is not necessary for verification or for the RAM load and execute modes. These other modes do, however, require at least 9 Volts on pin 2 immediately after reset. The charge pump will supply this voltage when S3 is opened if S4 has been left in position I.
5. Select required mode using switches S1 and S2.
6. Open S3; this starts the selected mode. The red LED flashes to indicate that the bootloader is running. At the end of the programming modes a verification is carried out and the green LED switched on if this verification passes. If the green LED is not on at the end of the procedure the verification has failed. The red LED may be on or off.
7. Close S3, at this stage the bootloader can be run again by returning to step 5.
8. If S4 is in position E, return it to position I. This disconnects an externally supplied Vpp. If this is not done before step 9, the MC68HC705T3 may be damaged.
9. Switch off Vdd.
10. Remove MC68HC705T3.
Figure 1. MC68HC705T3 Bootloader Circuit

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Program &amp; Verify HC05</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Verify HC05 Contents</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Load Program in RAM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Execute Program in RAM</td>
</tr>
</tbody>
</table>
ADDITIONAL FEATURES

A handshake facility has been included to allow the external EPROM to be replaced by an intelligent data source and to provide a limited debug capability. An alternative source of data could, for example, be a microprocessor controlling a gang programmer. When the direct connection shown in figure 1 between bits 0 and 1 on port C is made, the handshake is performed automatically. When this connection is not made the bootloader stops immediately before reading data from port B and outputs a high on bit 1. In order to proceed bit 0 should be pulsed high and low again. The high on bit 1 can be used to indicate that the MC68HC05T3 is ready to receive data.

If bit 0 is held low with a 10kohm resistor and a push-button switch connected between bit 0 and Vdd the bootloader can be single-stepped. This feature is intended for use with the RAM loader.

To effectively use single-stepping an LED should be connected to every port D pin. If the bootloader is started in the RAM loading mode with no link between bits 0 and 1 on port C, it will stop at the first address ($0100). Pressing the button will cause the bootloader to increment one address at a time (a debouncing circuit may be required to prevent it skipping addresses). The LEDs display addresses A0–A6. If switch S2 is closed once the bootloader has started, the LEDs display the MC68HC05T3’s RAM data instead of the address. With S2 closed the contents of the external EPROM are ignored. Although the loader does not permit the inspection of any locations other than $0100–$01FF, it is possible to read other locations using a program loaded into RAM. The data can be saved in unused locations between $0100 and $01FF and then read using the above procedure.

SOFTWARE

The bootloader resides in the area occupied by the selftest program in the ROMed MC68HC05T3. This area extends from $7F00 to $7FEF with the last 16 bytes ($7FE0–$7FEF) intended for vectors. In this bootloader only the reset vector is actually used.

The program starts at address $7F00 and immediately checks the state of bits 2 and 3 on port C. If they are both high a jump to $0100 is executed. This is to allow the running of a program previously loaded into RAM. If either I/O line is low, the software switches the OSD character EPROM into the memory map and initialises the ports. As the port B pins are always inputs, its data direction register does not need to be written to. The subroutine STXHIS is then used to initialise the address bus to the first EPROM location to be used ($0400). As the MC68HC705T3, in common with all M6805 microprocessors, has only an 8-bit index register, it is necessary to execute a program in RAM to allow the required flexibility in selecting the address for writing to, and reading from, the on-chip EPROM. For writing to EPROM this program consists simply of an extended STA instruction followed by a 2-byte address and an RTS instruction. This program is built by transferring the contents of the 6 bytes at TABLE into RAM. The 5th and 6th bytes are constants used for controlling EPROM write timing (actually 8 bytes are transferred but only the first six are relevant). This transfer is accomplished by the loop at MOVE. After this, a conditional branch depending on the level of bit 2 in port C is taken. If the line is high, the RAM loader is entered, but if it is low the EPROM bootloader modes (program or verify according to the level of bit 3) are started.

The RAM loader transfer the contents of $0100–$01FF of the external EPROM into the same addresses in the MC68HC05T3 and then verifies them, switching on the LED (bit 7 of port D) and entering the STOP mode if the verify succeeds. If the verify fails, the program will hang at the first address to fail. If during this process bit 3 of port C becomes high, then the external data is ignored and the contents of internal RAM appear, inverted, on port D. The decision to display data rather than write it is made by two conditional branches, one in the main loop of the RAM loader and one in the subroutine HAND1. This subroutine also performs handshaking and reads the external EPROM. If bit 3 of port C is high at the end of a successful verify, a jump to address $0100 is performed instead of a STOP.

The EPROM loader similarly loops round its required addresses ($0400–$0AFF, $2000–$7EFF and $7FF0–$7FFF), transferring data from external EPROM. For the EPROM, however, several milliseconds are required to write data. The bootloader performs two loops, each incorporating a 2mS write time per byte. After this is completed (it takes 110 seconds) a verify loop is performed and the green LED on bit 7 of port D is switched on if the verify succeeds. Alternatively, if an error is detected, the verify loop will hang up at the address of the first byte which did not verify. An inspection of the address bus (port A) will determine the lower address of this location. The high-order addresses can be seen between the output of the latch and the external EPROM. If all eight LEDs are fitted the high-order address will be displayed.

The last page of the listing consists of the subroutines STXHI which has two entry points, the subroutine NXTADR, TABLE and the bootloader’s vectors. NXTADR is responsible for incrementing the EPROM address and skipping the areas not required for EPROM programming.
This software was developed by Motorola Ltd. for demonstration purposes.
No liability can be accepted for its use in any specific application.
Original software copyright Motorola - all rights reserved.

P. Topping
14-Feb-91

PORTA EQU $00 Port A address
PORTB EQU $01 Port B
PORTC EQU $02 Port C
PORTD EQU $03 Port D
DDRA EQU $04 Port A data direction reg.
DDRC EQU $06 Port C
DDRD EQU $07 Port D
PROG EQU $1C EPROM program register
TR1 EQU $3E Test register 1

Switch options.
2,C 3,C

0 0 Program and verify EPROM.
0 1 Verify EPROM.
1 0 Parallel RAM load/verify.
1 1 Execute prog. in RAM.

Note: ROM always starts at $2000. Vectors are always at $7FFF. These are not the normal addresses for T1 or T2.

ORG $40
RAM RMB 1
ADDR RMB 2
RET RMB 1
LOOP RMB 1
TIME RMB 1

ORG $7F00
Execute program in RAM, port set-up, dump.

*******************************

START BRCLR 2, PORTC, BOOT CHECK FOR JMP TO RAM OR BOOT
START BRCLR 3, PORTC, BOOT
RSTRT JMP $0100 JUMP TO PROGRAM IN RAM

BOOT BSET 0, TR1 SWITCH OSD CHARACTER EPROM INTO MEMORY MAP
LDA #$FF
STA DDRA ALL OUTS, ADDRESSES
STA DORD ALL OUTS, LEDS
STA PORTD LEDS OFF
LDA #$00100010 0,1: HANDSHAKE, 2 & 3: OPTION SWITCHES.
STA DORC 4: TCP, 5: LATCH, 6 & 7: NOT THERE

CLRA
LDX $4 X <- 00000001
JSR STXHIS ADN1 <- $04, ADLO <- $00
LSIX X <- $08

MOVE LDA TABLE-1.X
STA RAM-1.X
DECX
BNE MOVE

START BRCLR 2, PORTC, PRGVER EPROM OR RAM

*******************************

Parallel RAM load/verify ($0100-$01FF).

*******************************

LDRA SM BSR RMSTRT HANDSHAKE AND GET DATA
LDRA SM BSR HAND1
PLOOP BRSET 3, PORTC, NEXT SKIP RAM LOAD?
NEXT INC PORTA
IN CX
BNE PLOOP

BRSET 3, PORTC, RSTRT IF PUT HIGH DURING VERIFY
BRA FIN THEN EXECUTE PROGRAM IN RAM.

BRSET 3, PORTC, RSTRT IF PUT HIGH DURING VERIFY
BRA FIN THEN EXECUTE PROGRAM IN RAM.

*******************************

717
0113  
0114  
0115  
0116  
0117  
0118  
0119  
0120  
0121  
0122  
0123  
0124  
0125  
0126  
0127  
0128  
0129  
0130  
0131  
0132  
0133  
0134  
0135  
0136  
0137  
0138  
0139  
0140  
0141  
0142  
0143  
0144  
0145  
0146  
0147  
0148  
0149  
0150  
0151  
0152  
0153  
0154  
0155  
0156  
0157  
0158  
0159  
0160  
0161  
0162  
0163  
0164  
0165  
0166  
0167  
0168  
0169  
0170  
0171  
0172  
0173  
0174  
0175  

0114 Program & verify EPROM.  
0115 $0400-$0AFF, $2000-$7EFF & $7FF0-$7FFF.  

********************
PRGVER BRSET 3,PORTC,VERF DO A VERIFY ONLY ?
********************
0120 7f52 06 02 22
0121
0122 7f55 ad 3c
PRGLOP BSR HAND2 HANDSHAKE AND GET DATA
0123 7f57 1a 1c
BSET 5,PROG LATCH ADDRESS & DATA
0124 7f59 bd 40
JSR RAM WRITE ONE BYTE
0125 7f5b 10 1c
BSET 0,PROG APPLY VPP
0126 7f5d b6 45
LDA TIME GET PROGRAMMING TIME IN ms
0127 7f5f ae a6
DELMS DEX 2ms-INNER LOOP (2MHz XTAL)
0128 7f61 5a
MS1 BNE x A OUTER LOOP
0129 7f62 26 fd
DECA x A OUTER LOOP
0130 7f64 4a
BNE DELMS REMOVE VPP
0131 7f65 26 f8
CLR PROG NEXT ADDRESS
0132 7f67 3f 1c
BSR NXTADR DONE ?
0133 7f69 ad 45
DECA x A OUTER LOOP
0134 7f6b 26 e8
BNE PRGLOP DO PROG LOOPS TWICE
0135
0136 7f6d ae 04
LDA $4 GET INITIAL MS ADDR
0137 7f6f bf 41
JSR STXHI A8-A14 <- $0400
0138 7f71 ad 32
BNE NXTADR ADDRH <- $0400
0139 7f73 3a 44
DECA x A OUTER LOOP
0140 7f75 26 de
BNE PRGLOP ADDRH <- $0400
0141
0142 7f77 3c 40
VERF INC RAM CHANGE STA TO EOR
0143 7f79 ad 18
CHECK BSR HAND2 HANDSHAKE AND GET DATA
0144 7f7b bd 40
JSR RAM COMPARE WITH EPROM BYTE
0145 7f7d 26 fe
HANG1 BNE HANG1 HANG UP IF DIFFERENT
0146 7f7f ad 2f
BNE NXTADR NEXT ADDRESS
0147 7f81 26 f6
BNE CHECK
0148
0149 7f83 a6 7f
FIN LDA $7F VERIFY LED ON, REST OFF
0150 7f85 b7 03
STA PORTD
0151 7f87 8e
STOP
0152
0153
0154
0155
0156
0157
0158
0159 7f88 9f
HAND1 TXA ADDRLO (RAM)
0160 7f8a a4 7f
AND $7F MAKE SURE VERIFY LED IS OFF
0161 7f8b 07 02 07
BRCLR 3,PORTC,DISP DISPLAY RAM DATA OR ADDRESS ?
0162 7f8e d6 01 00
LDA $0100,X DATA
0163 7f91 20 02
BRA DISP
0164
0165 7f99 b6 41
HAND2 LDA ADDR ADDRH (EPROM)
0166 7f9b 4f 43
DISP COMA
0157 7f9c b7 03
STA PORTD DISPLAY ADDRESS (OR DATA)
0168 7f9e 12 02
BSET 1,PORTC HANDSHAKE HIGH
0159 7f9a 01 02 fd
BCLR 0,PORTC,* AND LOW AGAIN
0170 7f9d 13 02
BSET 0,PORTC,*
0171 7f9f 00 02 fd
RS T READ AN EXTERNAL BYTE
0172
0173 7fa2 b6 01
0174 7fa4 81

718
Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.

Subroutines.
Additional Information
Additional Information

Additional information relevant to 8-bit MCU applications may be found in the following Motorola documents, available through your Franchised Distributor by quoting the appropriate reference.

- BR266/D M68HC11EVM Evaluation Module (Rev. 3)
- BR278/D M68HC11EVB Evaluation Board (Rev. 2)
- BR285/D M68701EVM Evaluation Module
- BR291/D M68705EVM Evaluation Module
- BR295/D M68HC05EVM Evaluation Module (Rev. 2)
- BR411/D The M68HC11 Microcontroller Family
- BR433/D M68HC05 8-bit Microcontrollers. The Home of the Industry Standard Microcontroller (Rev. 2)
- BR459/D M68HC05SC24 Secure 8-bit Microcomputer with EEPROM: Product Preview
- BR468/D Secure MCU Product Packaging
- BR568/D MCU Freeware (Rev. 1)
- BR706/D M68HC11F1EVM Evaluation Module
- BR730/D M68HC05PGMR Programmer Board
- BR735/D M68HC05P8EV5 CSIC Evaluation System
- BR736/D M68HC11EVBU Universal Evaluation Board
- BR748/D M68HC711D3PGMR Programmer Board
- BR784/D M68HC05 CSIC Portfolio
- BR909/D The Military Microprocessor Fleet is Arriving
- BR911/D Military Microprocessor Fact Sheet (Rev. 4, 1992)
- BR913/D The Military M68HC11A0 and M68HC11A1 are Available Now
- BR922/D Military MCU – 68HC811E2
- BR1111/D M68HC705J2/P9PGMR Programmer Board
- BR1133/D M68HC705B5PGMR Programmer Board
- BR1166/D Advanced Microcontroller Unit (AMCU) Literature
- BR1310/D Our Low-Cost 68HC05 CSICs Can Take Your Designs to New Heights
- BRE435/D M1468705EVM Evaluation Module (replaces BRE294/D)
- BRE447/D M6805SC13 Product Preview
- BRE448/D M68HC05SC1121 Product Preview
- DL411/D Communications Applications Manual
- M68HC05AG/AD M68HC05 Applications Guide
- M68HC05PGMR/AD1 M68HC05PGMR Programmer Board User's Manual
- MC68HC05CxRG/AD MC68HC05Cx HCMOS Single-Chip Microcontrollers Programming Reference Guide (Rev. 1)
- MC68HC11A8RG/AD MC68HC11A8 Programming Reference Guide (Rev. 1)
<table>
<thead>
<tr>
<th>Part Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC68HC11E9RG/AD</td>
<td>MC68HC11E9 Programming Reference Guide</td>
</tr>
<tr>
<td>MC68HC11F1RG/AD</td>
<td>MC68HC11F1 Programming Reference Guide</td>
</tr>
<tr>
<td>MC68HC11L6RG/AD</td>
<td>MC68HC711L6 Programming Reference Guide</td>
</tr>
<tr>
<td>MC6840UM/AD1</td>
<td>MC6840 Programmable Timer Fundamentals and Applications</td>
</tr>
<tr>
<td>SG165/D</td>
<td>CSIC Microcontrollers Update – Quarter 2, 1992</td>
</tr>
<tr>
<td>SG166/D</td>
<td>Advanced Microcontroller Division Update – Quarter 4, 1991</td>
</tr>
<tr>
<td>TB301/D</td>
<td>Basic Microprocessors and the 6800 (Bishop, 1979)</td>
</tr>
<tr>
<td>TB302/D</td>
<td>What Every Engineer Should Know About Microcomputers (Bennett, Evert and Lander, Rev. 1, 1991)</td>
</tr>
<tr>
<td>TB303/D</td>
<td>Using Microprocessors and Microcomputers: The Motorola Family (Greenfield and Wray, Rev. 1, 1988)</td>
</tr>
<tr>
<td>TB309/D</td>
<td>Programming the 6809 (Zaks &amp; Labia, 1982)</td>
</tr>
<tr>
<td>TB316/D</td>
<td>Single- &amp; Multi-Chip MCU Interfacing (Lipovski, 1988)</td>
</tr>
<tr>
<td>TOOLWARE/D</td>
<td>Software Development Tools for MS-DOS</td>
</tr>
</tbody>
</table>
Literature Distribution Centers:
USA: Motorola Literature Distribution; P.O. Box 20912; Phoenix, Arizona 85036.
EUROPE: Motorola Ltd.; European Literature Centre; 88 Tanners Drive, Blakelands, Milton Keynes, MK14 5BP, England.
JAPAN: Nippon Motorola Ltd.; 4-32-1, Nishi-Gotanda, Shinagawa-ku, Tokyo 141, Japan.
ASIA PACIFIC: Motorola Semiconductors H.K. Ltd.; Silicon Harbour Center, No. 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong.