AMI 6800 MICROCOMPUTER
CHIPS

By Robert A. Stevens
This article is the first one of a series of four articles covering AMI's S6800 microcomputer chip set. EVK Microcomputer Prototyping boards and EVK prototyping board PROTO & (RS)3 program development software.

The first article in this series covers the S6800 MPU in detail and summarizes the microcomputer supporting IC's in order to lay the ground work for next months article on AMI's EVK Prototyping boards.

AMI's S6800 FAMILY OF MICROCOMPUTER IC's

AMI's S6800 family of microcomputer IC's is composed of a series of matched MOS large scale integrated (LSI) circuits for configuring into microcomputer systems. This series of microcomputer MOS LSI functional building block logic circuits include a MPU, ROM, EPROM, RAM, PIA, ACIA, USRT, and Digital Modem Logic circuits.

S6800 — 8-BIT MICROPROCESSOR
FUNCTIONAL DESCRIPTION

S6800 MICROPROCESSOR (MPU) — an 8-bit parallel processor, with the ability to address up to 65K bytes of memory, and execute instructions in 2 micro-seconds. It is manufactured using N-channel MOS technology and operates on a single +5V power supply. All inputs and outputs are TTL compatible. The MPU has six internal registers, four types of vectored interrupts and 72 basic instructions. The basic instructions can be used in different addressing modes to save instruction execution time and memory space.

FEATURES

- Eight-Bit Parallel Processing
- Bi-Directional Data Bus
- Sixteen-Bit Address Bus — 65536 Bytes of Addressing
- 72 Instructions — Variable Length
- Seven Addressing Modes — Direct, Relative, Immediate, Indexed, Extended, Implied and Accumulator
- Variable Length Stack
- Vectored Restart
- 2 Microsecond Instruction Execution
- Maskable Interrupt Vector
- Separate Non-Maskable Interrupt — Internal Registers Saved in Stack

FIGURE 1. BLOCK DIAGRAM OF S6800 MICROPROCESSOR
• Six Internal Registers — Two Accumulators, Index Register, Program Counter, Stack Pointer and Condition Code Register
• Direct Memory Access (DMA) and Multiple Processor Capability
• Clock Rates as High as 1 MHz
• Simple Bus Interface Without TTL
• Halt and Single Instruction Execution Capability

S6800 MICROPROCESSOR ARCHITECTURE

The S6800 Microprocessor (MPU) is an 8-bit parallel processor. It contains an 8-bit arithmetic unit (ALU), two 8-bit accumulators, one condition code register, and three 16-bit address storage registers, all of which are available for program use (see Figure 1). In addition, there are the following non-accessible registers: a 16-bit address incrementer/decrementer, an 8-bit temporary register and an 8-bit instruction register. There is also an instruction decode ROM and cycle control logic, interrupt and restart logic, bus control and halt logic, and a timing generator.

MPU PROGRAM ACCESSIBLE REGISTERS

Accumulators A and B — Two separate 8-bit accumulators that are used to hold operands and results of operations in the ALU.

Index Register — A 16-bit register used for memory address storage in Indexed Addressing operations.

Program Counter — A 16-bit register that holds the current program instruction address. Once the initial program starting address is loaded into the program counter, it is incremented under control of the MPU hardware.

Stack Pointer — A 16-bit register used for storage of the next available location in an external push-down/pop-up stack.

Condition Code Register — An 8-bit register that stores certain results of operations in the ALU. These bits are used as testable conditions for the conditional branch instructions. In addition, one bit position stores the interrupt mask bit and the two high order bits are unused. See Figure 2.

MPU HARDWARE REGISTERS (NOT ACCESSIBLE BY PROGRAM)

Instruction Register — 8-bit register used to receive and store all program instructions input into the MPU (via the data bus lines D0-D7).

Temporary Register — 8-bit register typically used to store the high order address bits prior to their output from the MPU onto the external address bus lines A8-A15.

Incrementer — 16-bit auxiliary address register, used by the MPU internal control logic, in conjunction with the program counter, to maintain and output the current program address.

MPU INTERNAL BUSSES

Within the MPU all data and address transfers between the registers, as well as to and from the ALU, are made across three internal 8-bit busses. The first is a data bus, the second is an address bus for the low order bits, and the third is an address bus for high order bits.

MPU INTERFACE DESCRIPTION

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>(3)</td>
<td>Clocks Phase One and Phase Two — Two pins are used for a two-phase non-overlapping clock that runs at the Vcc voltage level.</td>
</tr>
<tr>
<td>02</td>
<td>(37)</td>
<td>RESET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reset — This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. If a positive edge is detected on the input, this will signal the MPU to begin the restart sequence. This will start execution of a routine to initialize the processor from its reset condition. All the higher order address lines will be forced high. For the restart, the last two (FFFE, FFFF) locations in memory will be used to load the program that is addressed by the program counter. During the restart routine, the interrupt mask bit is set and must be reset before the MPU can be interrupted by IRQ. Reset must be held low for at least eight clock periods after VCC reaches 4.75 volts (Figure 4). If Reset goes high prior to the leading edge of 02, on the next 01 the first restart memory vector address (FFFF) will appear on the address lines. This...</td>
</tr>
</tbody>
</table>

FIGURE 2.

EXTERNAL STACK MEMORY REGISTER — a push-down/pop-up stack that can be located anywhere in RAM and be of any convenient size. It is accessed with the stack pointer address and has several uses. First, it always stores the MPU register contents following an interrupt and return addresses during sub-routine execution. Second, it can also be used by the programmer to store data during program execution.
Three-State Control - This input causes the data bus to go into the off or high impedance state. This state will occur 500 ns after TSC = 2.4 V. The Valid Memory Address and Bus Available signals will be forced low. The data bus is not affected by TSC and has its own enable (Data Bus Enable). In DMA applications, the Three-State Control line should be brought high on the leading edge of the Phase One Clock. The 01 clock must be held in the high state and the 02 in the low state for this function to operate properly. The address bus will then be available for other devices to directly address memory. Since the MPU is a dynamic device, it can be held in this state for only 5.0 µs or destruction of data will occur in the MPU.

Bus Available - The Bus Available signal will normally be in the low state; when activated, it will go to the high state indicating that the microprocessor has stopped and that the address bus is available. This will occur if the Halt line is in the low state or the processor is in the WAIT state as a result of the execution of a WAIT instruction. At such time, all three-state output drivers will go to their off state and other outputs to their normally inactive level. The processor is removed from the WAIT state by the occurrence of a maskable (mask bit I = 0) or nonmaskable interrupt. This output is capable of driving one standard TTL load and 30 pF.

Interrupt Request - This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further interrupts may occur. At the end of the cycle, a 16-bit address will be loaded.

Valid Memory Address - This output indicates to peripheral devices that there is a valid address on the address bus. In normal operation, this signal should be utilized for enabling peripheral interfaces such as the PIA and ACIA. This signal is not three-state. One standard TTL load and 30 pF may be directly driven by this active high signal.

Address Bus - Sixteen pins are used for the address bus. The outputs are three-state bus drivers capable of driving one standard TTL load and 130 pF. When the output is turned off, it is essentially an open circuit. This permits the MPU to be used in DMA applications.

Available signals will be forced low. The data bus is not affected by TSC and has its own enable (Data Bus Enable). In DMA applications, the Three-State Control line should be brought high on the leading edge of the Phase One Clock. The 01 clock must be held in the high state and the 02 in the low state for this function to operate properly. The address bus will then be available for other devices to directly address memory. Since the MPU is a dynamic device, it can be held in this state for only 5.0 µs or destruction of data will occur in the MPU.
that points to a vectoring address which is located in memory locations FFF8 and FFF9. An address loaded at these locations causes the MPU to branch to an interrupt routine in memory.

The Halt line must be in the high state for interrupts to be recognized.

The IRQ has a high impedance pullup device internal to the chip; however a 3 kΩ external resistor to Vcc should be used for wire-OR and optimum control of interrupts.

**Non-Maskable Interrupt** — A low-going edge on this input requests that a non-mask interrupt sequence be generated within the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the NMI signal. The interrupt mask bit in the Condition Code Register has no effect on NMI.

The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away in the stack. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations FFFC and FFFD. An address loaded at these locations causes the MPU to branch to a non-maskable interrupt routine in memory.

NMI has a high impedance pullup resistor internal to the chip; however a 3 kΩ external resistor to Vcc should be used for wire-OR and optimum control of interrupts.

Inputs IRQ and NMI are hardware interrupt lines that are acknowledged during Ø2 and will start the interrupt routine on the Ø1 following the completion of an instruction.

**MPU EXTERNAL BUSSES**

The MPU communicates with its external memory and all I/O devices across an 8-bit bidirectional data bus, D0 through D7, and 16 address lines, A0 through A15. The MPU can be disconnected from either bus by two control signals DBE and TSC. In addition, a control bus maintains control of the bi directional Data Bus and provides access for control signals between the MPU and all external logic.

The MPU I/O bus relegates control to the programmed I/O devices, provides memory mapped I/O addressing and uses memory and register instructions to control all I/O operations. The MPU bus configuration is shown in Figure 3.

**Programmed I/O Devices** — The MPU relegates most of the I/O control to such I/O interfaces as the PIA or ACIA. Each of these circuits is programmable and can interface with peripheral devices without directly involving the MPU. For example, the MPU can preprogram a PIA to either output data to the MPU or to receive it. Thereafter, the PIA circuits assume all functions of interfacing with the peripherals and the MPU never has to look at the interface until service is required. It must service interrupts from the PIA, but never needs to wait for input data to become available or for output data to be accepted.

**MEMORY MAPPED I/O Addressing** — The I/O interfaces and memory are both located in the same address space within the S6800 system. The MPU can access any I/O device the same as a memory location.
— with address lines, instead of separate I/O control lines. Therefore, it can manipulate data in the I/O interface registers with the same programmed instructions as it uses for memory locations. This adds flexibility and increases system efficiency.

**No Special I/O Instructions** — The S6800 Instruction Set complements the above I/O addressing capability with specific instructions that can be used to access memory as well as I/O circuit registers and perform directly various manipulations on the data.

**BUS INTERFACE** — The bus interface consists of the Data Bus (D0-D7), the Address Bus (A0-A15), Read/Write (R/W), and Valid Memory Address (VMA). There are other signals which further control the operation of the MPU and thus affect the bus without actually being properly part of the bus interface; these include Data Bus Enable (DBE), Three-State Control (TSC), Halt and the interrupt control signals IRQ and NMI.

**DATA BUS** — The Data Bus comprises eight bidirectional data lines, which connect the MPU, all of the memory, and any I/O devices which may be addressed by the MPU. The MPU normally controls this bus during O2 time for the transfer of instructions and data into the MPU and the transfer of data out of the MPU. The direction of the data on the bus is a function of the R/W line generated by the MPU; a high on R/W constitutes a read, and the MPU accepts data during the latter portion of O2; a low on the R/W line is defined as a write out of the MPU, and the MPU drives the bus with the write data shortly after the low-to-high transition of DBE. Control of the Data Bus may be preempted from the MPU for Direct Memory Access (DMA) operations during O2 by operating the Halt line (Bus Available will go high when the bus is available for other than MPU-controlled use), or during O1 while DBE is low, and the MPU is not concerned with the contents of the data bus. When the MPU is not driving the data bus in a write operation, these lines are placed in a high impedance state to minimize the interference with other devices driving the bus; similarly when a memory or I/O device is not driving the bus in a read operation for which that device is selected, the bus lines in that device are placed in a high impedance state. At any one time only one device should be driving the bus, with all other connected system components in the high impedance state.

**Address Bus** — The Address bus comprises 16 address lines, by which the MPU identifies which of the 65,536 possible memory locations is to be read out or written into. In normal operation the MPU sets an address on the bus during O1 while TSC is low; this remains stable throughout O2 for the memory access operation. For DMA and other circumstances in which it is desired to control the Address Bus apart from the MPU, O1 may be extended during which time TSC may be set high; the MPU will respond by taking the Address Bus and R/W outputs to the high impedance state and outputting a low on VMA. A high on TSC also forces BA low.

The Valid Memory Address (VMA) output from the MPU should always be used in conjunction with the address on the Address Bus to determine whether the MPU is actually accessing memory (or peripheral registers), since in some circumstances the MPU will issue
a temporary address in a read or write cycle which might be interpreted as a “false read” or a “false write” to some memory location. VMA may be thought of as a 17th bit of address, where only half of the addressable memory (i.e. the VMA bit = 1) is usable, although occasionally the MPU will attempt to read or write some location in the other half (i.e. VMA = 0). Thus it can be combined with the higher order address bits to select or deselect memory and peripherals, as required by the MPU at that time.

**Bus Control Signals** — If the VMA signal is not used to deselect memory and I/O registers (PIAs and ACIAs), false reads or false writes may result in ambiguous operation. These are of particular concern in the case of RAM memories and the I/O devices register (PIAs and ACIAs), since in the case of the ROM any false reads are ignored by the MPU and have no other effect, and false writes have no effect on the ROM. In the case of the RAM a false read also is of no concern, but if TSC or Halt is used or the MPU executes a WAI instruction the R/W line floats in the high impedance state which could be interpreted as a write by the RAM; the TST instruction actually results in a false write, but the data and address are the same as an immediately preceding read, so the contents of RAM are not thus altered. For PIAs and ACIAs the VMA signal must be used in the selection logic, since a read from a PIA or ACIA register is used to clear an interrupt condition, and the failure to disable false reads could result in a missing interrupt. In the case of the PIA, VMA should not be used in the form of VMA-02 for the Enable (E) input, since at least one E pulse is required before each active transition of the CA1 (or CB1, etc.) to detect the interrupt; a WAI instruction depending on this transition may thus lock out interrupts by setting VMA low. It is better to apply VMA to one of the Chip Select inputs to the PIA (CSO, CS1, or inverted to CS2), or to specify the design to preclude the requirement that interrupts be detected on the trailing edge of a pulse.

Read/Write (R/W) is a control signal generated by the MPU to define the direction of the Data Bus. When low, the MPU is driving the Data Bus, and the selected memory or peripheral should accept the data written into it; when R/W is high, the selected memory or peripheral is being read into the MPU, and should be driving the bus. This control goes into the high impedance state when the Address Bus is disabled by TSC.

R/W is routed to the various memory and peripheral components as part of the system control. ROMs should be disabled when R/W is low, since they cannot be written into. RAMs and PIAs use the R/W signal to distinguish read and write operations. The ACIA has four internal registers, of which two are selected in the read operation, and two are selected for write operations, by the connection of R/W to the appropriate ACIA input.

**MPU Operating Cycle**

Instructions are executed within the MPU in incremental time periods (MPU cycles), each consisting of one 01 clock period and one 02 clock period. When the MPU is operating on a 1 MHz input clock, each MPU cycle is 1 microsecond long. It takes a minimum of two MPU cycles or 2 microseconds to execute a single word instruction.

During the 01 period the MPU typically outputs a memory address to access (fetch) one 8-bit program instruction or data byte and then, during 02, loads the byte into an internal register. During the next 01 period the MPU executes the associated internal operation with the ALU and the registers. With this fetch-execute sequence an instruction may be completed in only two MPU cycles, or may require as many as 12. While the MPU is executing successive cycles, it is also common for it to overlap functions. For example, during any given clock period the MPU may be executing one instruction in the ALU or registers; while at the same time a fetch is being performed with the address in the program counter.

**Program Control**

These internal operations of the MPU, as well as the output of address, data, and control signals, are all managed by the instruction decode and control logic. For example, to perform the execute part of any instruction, the control logic generates signals that cause the ALU to perform addition, subtraction, or some Boolean logic function. These signals can also cause the contents of one register to be transferred into another, a register to be simply incremented or decremented, or some other similar function to occur. Such ALU and register operations are used to execute all of the S6800 instructions.

**MPU Addressing Modes**

The S6800 eight-bit microprocessing unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. During the fetch part of any MPU cycle a memory address is required in order to access a particular location in the external memory. This address is normally stored in the program counter. The program counter is 16 bits wide and therefore, can address any one of a maximum of 65,536 bytes.

At the beginning of a program sequence the MPU is initialized and the beginning address is loaded into the program counter. From there on the program counter is incremented automatically, so that at the end of any instruction cycle it stores the next instruction address. If the program contains an instruction to branch or jump to a different memory location, the op code must be followed by two bytes which load the new address into the program counter. There are, however, addressing techniques with which the jump can be accomplished by fetching only one new address byte out of the memory. For example, if the destination of a branch is within 129 locations forward of 125 locations back of the current program counter contents, Relative Addressing can be used. In this mode only the op code and one signed 8-bit byte is fetched from the memory and is added to the program counter contents.

In Indexed Addressing, a single byte is added to the contents of the index register and the result is transferred into the program counter. Thus, the above addressing variations can be used to reduce the number of bytes that need be fetched to generate a new address. This reduces the number of MPU cycles and
speeds up program execution.

The various addressing modes can also be used in a similar manner to generate the source or destination addresses for data. MPU addressing modes are summarized in the following:

**INSTRUCTION FORMAT**

<table>
<thead>
<tr>
<th>ONE WORD INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE # 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TWO WORD INSTRUCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE # 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>THREE WORD INSTRUCTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE # 1</td>
</tr>
</tbody>
</table>

**ACCUMULATOR ADDRESSING (ACCX)**

| OP CODE |

A single byte instruction addressing operands only in accumulator A or accumulator B.

**IMPLIED ADDRESSING**

| OP CODE |

Single byte instruction where the operand address is implied by the instruction definition (i.e., Stack Pointer, Index Register or Condition Register).

**IMMEDIATE ADDRESSING**

| OP CODE | IMMEDIATE OPERAND |

Two or three byte instructions with an eight or sixteen bit operand respectively. For accumulator operations the eight bit operand is contained in the second byte of a two byte instruction. For Index Register operations (e.g. LDUX) sixteen bit operand is contained in the second and third byte of a three byte instruction.

**DIRECT ADDRESSING**

| OP CODE | ADDRESS 0-255 |

Two byte instructions with the address of the operand contained in the second byte of the instruction. This format allows direct addressing of operands within the first 256 memory locations.

**EXTENDED ADDRESSING**

| OP CODE | ADDRESS |

Three byte instructions with the higher eight bits of the operand address contained in the second byte and the lower eight bits of address contained in the third byte of the instruction. This format allows direct addressing of all 65,536 memory locations.

**INDEXED ADDRESSING**

| OP CODE | INDEX ADDRESS |

Two byte instructions where the 8 bit unsigned address contained in the second byte of the instruction is added to the sixteen bit Index Register resulting in a sixteen bit effective address. The effective address is stored in a temporary register and the contents of the Index Register are unchanged.

**RELATIVE ADDRESSING**

| OP CODE | RELATIVE ADDRESS |

Two byte instructions where the relative address contained in the second byte of the instruction is added to the sixteen bit program counter plus two. The relative address is interpreted as a two’s complement number allowing relative addressing within a range of −125 to +129 bytes of the present instruction.

**Interrupts.**

The S6800 MPU can be interrupted by any of several signals and program instructions, each of which initiates a different sequence in the MPU. Including the Reset signal, there are four interrupts — three hardware interrupts (signal lines connected to the MPU) and one software interrupt (SWI instruction). Each class of interrupt is described in the following:

**Nonmaskable Interrupt (NMI)** — initiated by a low-going signal on the NMI line to the MPU; always interrupts the MPU — even while another interrupt is being processed and the interrupt mask bit is set. Therefore, NMI can be considered to the highest priority interrupt. It causes the following sequence of events:

1. At the completion of the instruction being executed, the contents of the program accessible registers (Figure 3) are stored in the stack.
2. The interrupt mask bit is set.
3. Starting with its next cycle, the MPU accesses locations FFFC and FFFD in the memory and loads the contents into the program counter.
Interrupt Request (IRQ) — initiated by a logic low signal on the IRQ line; interrupts the MPU as long as the interrupt mask bit is not set. It causes the following sequence of events:

1. At the completion of the instruction being executed, the interrupt mask bit is tested. If the bit is set the interrupt must wait; if it is not set, contents of the program accessible registers are stored in the stack.
2. The interrupt mask bit is set.
3. Starting with the next cycle, the MPU accesses locations FFFA and FFFB in the memory and loads the contents into the program counter.

Software Interrupt (SWI) — initiated by the SWI instruction and causes the following sequence of events:

1. Contents of the program accessible registers are stored in the stack.
2. The interrupt mask bit is set.
3. Starting with the next cycle, the MPU accesses locations FFFE and FFFF in the memory and loads the contents into the program counter.

Reset — initiated by a positive going edge on the RESET line to the MPU. It causes the following sequence of events:

1. All program accessible registers are cleared and other circuits in the MPU are initialized.
2. The interrupt mask bit is set.
3. Starting with the next cycle, the MPU accesses locations FFFE and FFFF in the memory and loads the contents into the program counter.

Wait (WAI) — an instruction that causes the MPU to stop all processing and wait for a hardware interrupt. This instruction is not an interrupt in itself because it does not cause branching to any memory address, however, it does cause contents of the program accessible registers to be stored into the stack, in preparation for an interrupt.

All interrupts are vectored — they cause the MPU to automatically access a predetermined location in the memory and fetch a branch address of the routine or program to which the MPU is to go to service the interrupt. All interrupts except Reset also cause the contents of each program accessible MPU register (with the exception of the stack pointer) to be transferred to the external stack and thus be saved for later processing. The IRQ interrupt is also maskable — it cannot interrupt the MPU as long as bit 4 in the condition code register is set.

The S6800 requires a 16-bit vector address to indicate the location of routines for Restart, Non-maskable Interrupt, and Maskable Interrupt. Additionally an address is required for the Software Interrupt Instruction (SWI). The processor assumes the uppermost eight memory locations, FFF8 — FFFF, are assigned as interrupt vector addresses as defined in Figure 4.

After completing the current instruction execution the processor checks for an allowable interrupt request via the IRQ or NMI inputs as shown by the simplified flow chart in Figure 5.

**FIGURE 4. MEMORY MAP FOR INTERRUPT VECTORS**

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFE</td>
<td>Restart</td>
</tr>
<tr>
<td>FFFC</td>
<td>Non-maskable Interrupt</td>
</tr>
<tr>
<td>FFFA</td>
<td>Software Interrupt</td>
</tr>
<tr>
<td>FFFB</td>
<td>Interrupt Request</td>
</tr>
</tbody>
</table>

**FIGURE 5. MPU FLOW CHART**

**FIGURE 6. SAVING THE STATUS OF THE MICROPROCESSOR IN THE STACK**

SP = Stack Pointer  
CC = Condition Codes (Also called the Processor Status Byte)  
ACCB = Accumulator B  
ACCA = Accumulator A  
IXH = Index Register, Higher Order 8 Bits  
IXL = Index Register, Lower Order 8 Bits  
PCH = Program Counter, Higher Order 8 Bits  
PCL = Program Counter, Lower Order 8 Bits

42 INTERFACE AGE  
DECEMBER 1976
Recognition of either external interrupt request or a Wait for Interrupt (WAI) or Software Interrupt (SWI) instruction causes the contents of the Index Register, Program Counter, Accumulators and Condition Code Register to be transferred to the stack as shown in Figure 6.

S6800 INSTRUCTION SET

The S6800 MPU has a set of 72 basic instructions, listed in alphabetical order in Table 1. These include binary and decimal arithmetic functions, as well as logical, shift, rotate, load, store, branch, interrupt, and stack manipulation functions. Most of the instructions have several variations and most can be used with several memory addressing modes. Thus, the total complex of instructions available to the programmer actually is 197.

An instruction can be from one to three bytes long, depending on the addressing mode used with the instruction. The first byte always contains the operation code, which designates the kind of operation the MPU will perform. In single byte instructions no memory address is required, because the operation is performed on one of the internal MPU registers. In multiple byte instructions the second and third byte can be the operand, or a memory address for the operand.

A noteworthy feature of the S6800 MPU is that some of the instructions can operate directly on any memory location. In other computer systems it is common that the processor fetches an operand from memory, stores it in the accumulator, then executes

<table>
<thead>
<tr>
<th>ABA</th>
<th>Add Accumulators</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
</tr>
<tr>
<td>AND</td>
<td>Logical And</td>
</tr>
<tr>
<td>ASL</td>
<td>Arithmetic Shift Left</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
</tr>
<tr>
<td>BCC</td>
<td>Branch if Carry Clear</td>
</tr>
<tr>
<td>BCS</td>
<td>Branch if Carry Set</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch if Equal to Zero</td>
</tr>
<tr>
<td>BGE</td>
<td>Branch if Greater or Equal Zero</td>
</tr>
<tr>
<td>BGT</td>
<td>Branch if Greater than Zero</td>
</tr>
<tr>
<td>BHI</td>
<td>Branch if Higher</td>
</tr>
<tr>
<td>BIT</td>
<td>Bit Test</td>
</tr>
<tr>
<td>BLE</td>
<td>Branch if Less or Equal</td>
</tr>
<tr>
<td>BLS</td>
<td>Branch if Lower or Same</td>
</tr>
<tr>
<td>BLT</td>
<td>Branch if Less than Zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Branch if Minus</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch if Not Equal to Zero</td>
</tr>
<tr>
<td>BPL</td>
<td>Branch if Plus</td>
</tr>
<tr>
<td>BRA</td>
<td>Branch Always</td>
</tr>
<tr>
<td>BSR</td>
<td>Branch to Subroutine</td>
</tr>
<tr>
<td>BVC</td>
<td>Branch if Overflow Clear</td>
</tr>
<tr>
<td>BVS</td>
<td>Branch if Overflow Set</td>
</tr>
<tr>
<td>CBA</td>
<td>Compare Accumulators</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear Carry</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear Interrupt Mask</td>
</tr>
<tr>
<td>CLR</td>
<td>Clear</td>
</tr>
<tr>
<td>CLV</td>
<td>Clear Overflow</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
</tr>
<tr>
<td>COM</td>
<td>Complement</td>
</tr>
<tr>
<td>CPX</td>
<td>Compare Index Register</td>
</tr>
<tr>
<td>DAA</td>
<td>Decimal Adjust</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement</td>
</tr>
<tr>
<td>DES</td>
<td>Decrement Stack Pointer</td>
</tr>
<tr>
<td>DEX</td>
<td>Decrement Index Register</td>
</tr>
<tr>
<td>FOR</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>INC</td>
<td>Increment</td>
</tr>
<tr>
<td>INS</td>
<td>Increment Stack Pointer</td>
</tr>
<tr>
<td>INX</td>
<td>Increment Index Register</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to Subroutine</td>
</tr>
<tr>
<td>LDA</td>
<td>Load Accumulator</td>
</tr>
<tr>
<td>LDS</td>
<td>Load Stack Pointer</td>
</tr>
<tr>
<td>LDX</td>
<td>Load Index Register</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical Shift Right</td>
</tr>
<tr>
<td>NEG</td>
<td>Negaate</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>ORA</td>
<td>Inclusive OR Accumulator</td>
</tr>
<tr>
<td>PSH</td>
<td>Push Data</td>
</tr>
<tr>
<td>PUL</td>
<td>Pull Data</td>
</tr>
<tr>
<td>ROL</td>
<td>Rotate Left</td>
</tr>
<tr>
<td>RDR</td>
<td>Rotate Right</td>
</tr>
<tr>
<td>RTI</td>
<td>Return from Interrupt</td>
</tr>
<tr>
<td>RTS</td>
<td>Return from Subroutine</td>
</tr>
<tr>
<td>SBA</td>
<td>Subtract Accumulators</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with Carry</td>
</tr>
<tr>
<td>SEC</td>
<td>Set Carry</td>
</tr>
<tr>
<td>SEI</td>
<td>Set Interrupt Mask</td>
</tr>
<tr>
<td>SEV</td>
<td>Set Overflow</td>
</tr>
<tr>
<td>STA</td>
<td>Store Accumulator</td>
</tr>
<tr>
<td>STS</td>
<td>Store Stack Register</td>
</tr>
<tr>
<td>STX</td>
<td>Store Index Register</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>SWI</td>
<td>Software Interrupt</td>
</tr>
<tr>
<td>TAB</td>
<td>Transfer Accumulators</td>
</tr>
<tr>
<td>TAP</td>
<td>Transfer Accumulators to Condition Code Reg.</td>
</tr>
<tr>
<td>TBA</td>
<td>Transfer Accumulators</td>
</tr>
<tr>
<td>TPA</td>
<td>Transfer Condition Code Reg. to Accumulator</td>
</tr>
<tr>
<td>TST</td>
<td>Test</td>
</tr>
<tr>
<td>TSX</td>
<td>Transfer Stack Pointer to Index Register</td>
</tr>
<tr>
<td>TXS</td>
<td>Transfer Index Register to Stack Pointer</td>
</tr>
<tr>
<td>WAI</td>
<td>Wait for Interrupt</td>
</tr>
</tbody>
</table>

TABLE 1. S6800 MICROPROCESSOR INSTRUCTION SET

DECEMBER 1976

INTERFACE AGE 43
the operation in the ALU, and finally writes the result back into the memory. The S6800 is able to accomplish the same with only a single instruction, because it operates with any external location in the same manner as with an internal register. For example, it can directly increment or decrement the contents of a memory location. Because the MPU addresses I/O devices just like a memory location, it can do the same with registers inside the PIA or ACIA. The ASL, ASR, LSR, and ROL are other examples of instructions which operate in this manner.

**S6810 — 128 X 8 STATIC READ/WRITE MEMORY**

**FUNCTIONAL DESCRIPTION**

The S6810 is a static 128 X 8 Read/Write Memory designed and organized to be compatible with the S6800 Microprocessor. Interfacing to the S6810 consists of an 8 Bit Bidirectional Data Bus, Seven Address Lines, a single Read/Write Control line, and six Chip Enable lines, four negative and two positive.

For ease of use, the S6810 is a totally static memory requiring no clocks or cell refresh. The S6810 is fabricated with N channel silicon gate technology to be fully DTL/TTL compatible with only a single +5 volt power supply required. See Figure 7 for Functional Block Diagram.

**S6820 — PERIPHERAL INTERFACE ADAPTER (PIA)**

**FUNCTIONAL DESCRIPTION**

The S6820 Peripheral Interface Adapter provides the universal means of interfacing peripheral equipment to the S6800 Microprocessing Unit (MPU). This device is capable of interfacing the MPU to peripherals through two I/O 8-bit bidirectional peripheral data buses and four control lines. No external logic is required for interfacing to most peripheral devices.

The functional configuration of the PIA is programmed by the MPU during system initialization. Each of the peripheral data lines can be programmed to act as an input or output, and each of the four control/interrupt request lines may be programmed for one of several control modes. This allows a high degree of flexibility in the over-all operation of the interface.

The PIA interfaces to the S6800 MPU with an eight-bit bidirectional data bus, three chip select lines, two register select lines, two interrupt request lines, read/write line, enable line, and reset line. These signals, in conjunction with the S6800 VMA output, permit the MPU to have complete control over the PIA. VMA may be utilized to gate the input signals to the PIA. See Figure 8 for Functional Block Diagram.
FEATURES
- 8-Bit Bidirectional Data Bus for Communication with the MPU
- Two Bidirectional 8-Bit Buses for Interface to Peripherals
- Two Programmable Control Registers
- Two Programmable Data Direction Registers
- Four Individually-Controlled Interrupt Input Lines; Two Usable as Peripheral Control Outputs
- Handshake Control Logic for Input and Output Peripheral Operation
- High-Impedance 3-State and Direct Transistor Drive Peripheral Lines
- Program Controlled Interrupt and Interrupt Disable Capability

S6830 — 1024 X 8 READ ONLY MEMORY

FUNCTIONAL DESCRIPTION
The S6830 is a mask programmable read only memory organized 1024 words x 8 bits for application in byte organized systems. The S6830 is totally bus compatible with the S6800 microprocessor. Interfacing to the S6830 consists of an 8 bit three-state data bus, four mask programmable chip selects and ten address lines.

The S6830 is a totally static memory requiring no clocks. Access time is compatible with maximum data rates in a S6800 microprocessor system. The device operates from a single +5 volt power supply and is fabricated with N-channel silicon gate technology. See Figure 9 for Function Block Diagram.

FIGURE 9. FUNCTIONAL BLOCK DIAGRAM

S6831/A/B/C — 2048 X 8 READ ONLY MEMORY

FUNCTIONAL DESCRIPTION
The S6831/A/B/C is a 16,384 bit mask programmable MOS Read Only Memory organized 2K words x 8 bits. This ROM has been designed to supply large bit storage, high performance memory for microprocessors and other demanding applications with simple interface requirements. The device will operate from a single +5V supply and is manufactured with a N-channel silicon gate depletion load technology. This device is available in all common high density ROM pinouts. See Figure 10 for Functional Block Diagram.

FIGURE 10. FUNCTIONAL BLOCK DIAGRAM

FEATURES
- Organized as 1024-Bytes of 8 Bits
- Static Operation
- Three-State Data Output
- Four Chip Enable Inputs (Mask Programmable)
- Single 5-Volt Power Supply
- TTL Compatible Input/Output
- Maximum Access Time = 575 ns
FEATURES

- Mask programmable
- Maximum Access Time = 450 ns@CL = 130 pF
- Low Power 150 mW avg.
- Organized as 2048-Bytes of 8 Bits
- Static Operation
- Three-State Data Output
- 3 Chip Enable Inputs (Mask Programmable)
- The S6831 is pinout similar with the S6830
- The S6831A is pinout compatible with the 2316A, 8316A
- The S6831B is pinout compatible with the Intel 2316B, MC68317
- The S6831C is pinout compatible with the EA4600
- Single 5-Volt Power Supply
- TTL Compatible Input/Output

S6834 — 512 X 8 BIT EPROM

FUNCTIONAL DESCRIPTION

The S6834 is a high speed, static, 512 x 8 bit, erasable and electrically programmable read only memory designed for the in bus-organized systems. Both input and output are TTL compatible during both read and write modes. Packaged in a 24 pin hermetically sealed dual in-line package the bit pattern can be erased by exposing the chip to an ultra-violet light source through the transparent lid, after which a new pattern can be written. See Figure 11 for Functional Block Diagram.

S6850 — ASYNCHRONOUS COMMUNICATION INTERFACE ADAPTER (ACIA)

FUNCTIONAL DESCRIPTION

The S6850 Asynchronous Communications Interface Adapter (ACIA) provides the data formatting and control to interface serial asynchronous data communications to bus organized systems such as the S6800 Microprocessing Unit.

The S6850 includes select enable, read/write, interrupt and bus interface logic to allow data transfer over an eight bit bi-directional data bus. The parallel data of the bus system is serially transmitted and received by the asynchronous data interface, with proper formatting and error checking. The functional configuration of the ACIA is programmed via the data bus during system initialization. Word lengths, clock division ratios and transmit control through the Request to Send output may be programmed. For modem operation three control lines are provided. These lines allow the ACIA to interface directly with the S6860 0-600 bps digital modem. See Figure 12 for Functional Block Diagram.

FIGURE 11. FUNCTIONAL BLOCK DIAGRAM

FEATURES

- On-Board Programmability
- Fast Access Time — 575 ms Max.
- Pin Configuration Similar to the S6830 1K x 8 Bit ROM
- High Speed Programming — Less than 1 Minute for all 4096 Bits
- Programmed with R/W, CS and VPROG Pins
- Completely TTL Compatible — Excluding the VPROG Pin

FIGURE 12. FUNCTIONAL BLOCK DIAGRAM

FEATURES

- Eight and nine-bit transmission with optional even and odd parity.
- Parity, overrun and framing error checking.
• Programmable control register.
• Optional -1, +16, and +64 clock modes.
• Up to 500,000 bps transmission.
• 8 Bit Bidirectional Data Bus for Communication with MPU.
• False start bit deletion.
• Peripheral/modem control functions.
• Double buffered Receiver and Transmitter.
• One or two stop bit operation.

S2350 — UNIVERSAL SYNCHRONOUS RECEIVER/TRANSMITTER (USRT)

FUNCTIONAL DESCRIPTION

The S2350 Universal Synchronous Receiver Transmitter (USRT) is a single chip MOS/LSI device that totally replaces the serial to parallel and parallel to serial conversion logic required to interface a word parallel controller or data terminal to a bit-serial, synchronous communication network.

The USRT consists of separate receiver and transmitter sections with independent clocks, data lines and status. Common with the transmitter and receiver are word length and parity mode. Data is transmitted and received in a NRZ format at a rate equal to the respective input clock frequency.

Data messages are transmitted as a contiguous character stream, bit synchronous with respect to a clock and character synchronous with respect to framing or "sync" characters initializing each message. The USRT receiver compares the contents of the internal Receiver Sync Register with the incoming data stream in a bit transparent mode. When a compare is made, the receiver becomes character synchronous formatting a 5, 6, 7, or 8 bit character for output each character time. The receiver has an output buffer register allowing a full character time to transfer the data out. The receiver status outputs indicate received data available (RDA), receiver overrun (ROR), receive parity error (RPE) and sync character received (SCR). Status bits are available on individual output lines and can also be multiplexed onto the output data lines for bus organized systems. The data lines have tri-state outputs.

The USRT transmitter outputs 5, 6, 7, or 8 bit characters with correct parity at the transmitter serial output (TSO). The transmitter is buffered to allow a full character time to respond to a transmitter buffer empty (TBMT) request for data. Data is transmitted in a NRZ format changing on the positive transition of the transmitter clock (TCP). The character in the transmitter fill register is inserted into the data message if a data character is not loaded into the transmitter after a TBMT request. See Figure 13 for Functional Block Diagram.

FEATURES
• 500 KHz Data Rates
• Internal Sync Detection
• Fill Character Register
• Double Buffered Input/Output
• Bus Oriented Outputs
• 5-8 Bit Characters
• Odd/Even or No Parity
S6860 — 0-600 BPS DIGITAL MODEM

FUNCTIONAL DESCRIPTION

The S6860 is a 0-600 bps Digital Modem circuit designed to be integrated into a wide range of equipment utilizing serial data communications.

The modem provides the necessary modulation, demodulation and supervisory control functions to implement a serial data communications link, over a voice grade channel, utilizing frequency shift keying (FSK) at bit rates up to 600 bps. The S6860 can be implemented into a wide range of data handling systems, including stand alone modems, data storage devices, remote data communication terminals and I/O interfaces for minicomputers.

N-channel silicon gate technology permits the S6860 to operate using a single voltage supply and be fully TTL compatible.

The modem is compatible with the S6800 microcomputer family, interfacing directly with the Asynchronous Communications Interface Adapter (ACIA) to provide low-speed data communications capability. See Figure 14 for Functional Block Diagram.

FEATURES

- TTL compatible terminal interfaces
- Crystal/External reference control
- Compatible functions for 100 series data sets and 1001 A/B data couplers
- Full or half duplex operation
- Originates and answers mode
- Auto answer and disconnect
- Modem self test

TYPICAL S6800 MICROCOMPUTER CONFIGURATION

The S6800 microcomputer functional IC components may be assembled in a modular building block manner into a very simple microcomputer system, or into any of progressively more complex systems, which can be used in many general or special purpose applications. The important feature of the S6800 family is that all microcomputer system components are directly compatible in signal functions, circuit performance characteristics, and logic levels. All operate on a single +5 Volt power supply.

A basic microcomputer system built with the S6800 functional components is shown in Figure 15. This basic microcomputer configuration includes a S6800 Microprocessor (MPU), 1K bytes of ROM program storage, 128 bytes of RAM working storage and a two part input/output peripheral interface circuit.

Two-Phase Clock Circuitry and Timing — The MPU requires a two-phase non-overlapping clock which has a frequency range as high as 1 MHz. In addition to the two phases, this circuit should also generate an enable signal E, and its complement E, to enable ROMs, RAMs, PIAs and ACIAs. This Enable signal and its complement is obtained by ANDing O2 and VMA (Valid Memory Address).
FIGURE 15. MINIMUM MICROCOMPUTER SYSTEM CONFIGURATION
Chip Selection and Addressing — The minimum system configuration permits direct selection of the ROM, RAM, ACIA and PIA without the use of special TTL select logic. This is accomplished by simply wiring the address lines A13 and A14 to the Enable or chip select lines on the memories and PIA. This permits the devices to be addressed as follows:

<table>
<thead>
<tr>
<th>DEVICE</th>
<th>A14</th>
<th>A13</th>
<th>HEX ADDRESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>0</td>
<td>0</td>
<td>0000–007F</td>
</tr>
<tr>
<td>PIA</td>
<td>0</td>
<td>1</td>
<td>2004–2007 (Registers)</td>
</tr>
<tr>
<td>ROM</td>
<td>1</td>
<td>1</td>
<td>6000–63FF</td>
</tr>
</tbody>
</table>

Other addressing schemes can be utilized which use any combination of two of the lines A10 through A14 for chip selection.

Peripheral Control — All control and timing for the peripherals that are connected to the PIA is accomplished by software routines under the control of the MPU.

Restart and Non-Maskable Interrupt — Since this basic system does not have a nonvolatile RAM, special circuitry to handle loss of power using NMI is not required. Circuitry is, however, required to insure proper initialization of the MPU when power is turned on. This circuit should insure that the Restart signal is held low for eight 01 clock cycles after the Vcc power supply reaches a voltage of approximately 4.75 volts DC. Also, in order to insure that a PIA or ACIA is not inadvertently selected during the power-on sequence, Three-State Control (TSC) should be held high until the positive transition of Restart.

HALT — The Halt line is tied to Vcc and will automatically place the MPU in the run state when power is turned on. This signal may be used to halt the MPU if a switch is used to tie the line to ground for HALT and to Vcc for RUN.

The basic microcomputer system can be altered or expanded on in many different ways. For example, the S6850 Asynchronous Communication Interface Adapter (ACIA) can be substituted for a PIA, to enable the microcomputer to interface with a telecommunications modem. Or, additional memory can be added — either RAM or ROM — to expand the processing capability of the MPU. In general, the system can be expanded in a modular manner, by adding onto the bus as many as ten devices out of the S6800 family of modules. These additional modules can be any combination of memory or I/O IC circuits. In this manner a system of nearly any complexity and configuration can be assembled. Microcomputer system configurations requiring more than ten devices on the MPU bus require the addition of address and data bus buffers to operate at full speed.

By building your microcomputer from the S6800 family of devices, you take full advantage of the compatibility of the devices. They all conform to the MPU bus discipline, all are compatible in load levels, and the entire system runs on a common system clock. In effect you eliminate most all circuit design, save for the simple clock and power-up restart circuits. Because you are dealing with only a small number of integrated circuits, PCB circuit layout is simple and the entire microcomputer can be located on a single small circuit card.

ARTICLE BACKGROUND MATERIAL

The majority of the material for this article was gleaned from AMI's excellent documentation with the intent of not redoing good work just for the sake of it. Hopefully I have organized this material and clarified it to the extent of making it clearer and easier to understand which was the intent.

Next month I will cover the hardware mechanization of the AMI, EVK Microcomputer Prototyping boards.

Want more information on AMI's microcomputer chip set? Write or call:
American Microsystems, Inc.
3800 Homestead Road
Santa Clara, Calif. 95051
Phone (408) 246-0330
AMI's EVK Series
Microcomputer
Prototyping Boards

By Robert A. Stevens
INTRODUCTION

This article is part 2 of a series of articles on the EVK Microcomputer hardware, firmware and supporting software. This month’s article covers the EVK Microcomputer board architecture while last month’s article described the functional architecture and characteristics of AMI’s Microcomputer IC chip set.

EVK CONFIGURATIONS — The AMI EVK Microcomputer is a single board microcomputer mechanized with a standard S6800 MPU. The EVK Microcomputer comes in four basic configurations: EVK99, EVK100, EVK200, & EVK300, all of which use the same 10½” x 12” printed circuit board. EVK99 is a kit that includes the PCB and Microcomputer ICs consisting of one 6800 MPU, four 6810 RAM’s, one 6820 PIA, two 6830 ROM, and one 6850 ACIA. EVK100 & EVK200 are kit configurations that include PCB, Microcomputer & T² IC’s and differ from each other by the amount of hardware, memory and firmware (software in ROM) included with each configuration. EVK300 is the EVK200 kit with more EPROM memory and is factory assembled and tested. A Tiny BASIC Interpreter program is also available at no charge for the EVK300 Microcomputer board. Table 1. EVK Microcomputer Configuration Summary, shows the comparison between the different EVK configurations.

<table>
<thead>
<tr>
<th>EVK BOARD CHARACTERISTICS</th>
<th>EVK 99</th>
<th>EVK 100</th>
<th>EVK 200</th>
<th>EVK 300</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>S6800</td>
<td>S6800</td>
<td>S6800</td>
<td>S6800</td>
</tr>
<tr>
<td>WORD SIZE</td>
<td>8 BITS</td>
<td>8 BITS</td>
<td>8 BITS</td>
<td>8 BITS</td>
</tr>
<tr>
<td>ADDRESS BUS</td>
<td>16 BITS (64K)</td>
<td>16 BITS (64K)</td>
<td>16 BITS (64K)</td>
<td>16 BITS (64K)</td>
</tr>
<tr>
<td>ROM</td>
<td>2K BYTES S6831</td>
<td>2K BYTES S6831</td>
<td>2K BYTES S6831</td>
<td>2K BYTES S6831</td>
</tr>
<tr>
<td>EPROM — VIRGIN</td>
<td></td>
<td></td>
<td>512 BYTES S6834</td>
<td>2K BYTES S6834</td>
</tr>
<tr>
<td>STATIC RAM</td>
<td>512 BYTES S6810</td>
<td>512 BYTES S6810</td>
<td>1K BYTES S6810</td>
<td>1K BYTES S6810</td>
</tr>
<tr>
<td>EPROM PROGRAMMING</td>
<td></td>
<td></td>
<td>PROGRAMS S6834 EPROM's</td>
<td>PROGRAMS S6834 EPROM's</td>
</tr>
<tr>
<td>I/O PORTS</td>
<td>1 PIA=2 PORTS – 8 BITS/PORT</td>
<td>3 PIA=6 PORTS 8 BITS/PORT</td>
<td>3 PIA=6 PORTS 8 BITS/PORT</td>
<td></td>
</tr>
<tr>
<td>ASR 33/35 TTY SERIAL INTERFACE</td>
<td>ACIA 56850</td>
<td>ACIA 56850 WITH 20mA CURRENT LOOP</td>
<td>ACIA 56850 WITH 20mA CURRENT LOOP</td>
<td>ACIA 56850 WITH 20mA CURRENT LOOP</td>
</tr>
<tr>
<td>RS232C EIA SERIAL INTERFACE</td>
<td>ACIA 56850</td>
<td>ACIA 56800</td>
<td>ACIA 56850 WITH EIA RS232C</td>
<td>ACIA 56850 WITH EIA RS232C</td>
</tr>
<tr>
<td>INTERVAL TIMER (CRYSTAL)</td>
<td></td>
<td></td>
<td>1 ms &amp; 100 μs TIME INTERVALS</td>
<td>1 ms &amp; 100 μs TIME INTERVALS</td>
</tr>
<tr>
<td>MPU CRYSTAL CLOCK</td>
<td></td>
<td></td>
<td>INCLUDED</td>
<td>INCLUDED</td>
</tr>
<tr>
<td>CLOCK OUTPUTS (CRYSTAL)</td>
<td></td>
<td>16X BAUD RATE</td>
<td>2.4576 MHz, 1 MHz &amp; 16X BAUD RATE</td>
<td>2.4576 MHz, 1 MHz &amp; 16X BAUD RATE</td>
</tr>
<tr>
<td>DMA MODES</td>
<td></td>
<td></td>
<td>HALT MPU MODE, CYCLE STEAL MODE &amp; MUX MODE</td>
<td>HALT MPU MODE, CYCLE STEAL MODE &amp; MUX MODE</td>
</tr>
<tr>
<td>RESTART ADDRESS SELECTION</td>
<td></td>
<td>TWO 8 BIT DIP TOGGLE SWITCHES</td>
<td>TWO 8 BIT DIP TOGGLE SWITCHES</td>
<td>TWO 8 BIT DIP TOGGLE SWITCHES</td>
</tr>
<tr>
<td>TTY MONITOR SOFTWARE</td>
<td>PROTO ROM RESIDENT</td>
<td>PROTO ROM RESIDENT</td>
<td>PROTO ROM RESIDENT</td>
<td>PROTO ROM RESIDENT</td>
</tr>
<tr>
<td>SUBROUTINE PROGRAM LIBRARY SOFTWARE</td>
<td>P3 ROM RESIDENT</td>
<td>P3 ROM RESIDENT</td>
<td>P3 ROM RESIDENT</td>
<td>P3 ROM RESIDENT</td>
</tr>
<tr>
<td>ROM RESIDENT ASSEMBLER</td>
<td>$350 OPTION</td>
<td>$350 OPTION</td>
<td>$350 OPTION</td>
<td>$350 OPTION</td>
</tr>
<tr>
<td>OEM SINGLE QUANTITY PRICE</td>
<td>$135.00</td>
<td>$295.00</td>
<td>$405.00</td>
<td>$565.00</td>
</tr>
</tbody>
</table>

TABLE 1 EVK MICROCOMPUTER CONFIGURATION SUMMARY
MAJOR EVK MICROCOMPUTER FEATURES
The common denominator EVK Microcomputer PCB provides the following on board major features when fully populated with hardware and software including options:
• 4K Bytes S6831 ROM memory (2K using S6831 ROM's)
• 2K Bytes S6834 EPROM memory
• 1 K Bytes S6810 Static RAM memory
• On board S6834 EPROM programming
• Six 8 bit PTA I/O ports
• 20 ma serial TTY current loop port interface
• RS232C EIA serial I/O port
• Switch selected baud rates to 19,200 bauds
• 1 MHz crystal or variable one shot MPU clock
• 5 crystal controlled timing signals available at PCB interface (2.4756 MHz, 1 MHz, 16x baud rate, 100 µs & 1 ms)
• Interrupt internal timing (100µs & 1 ms)
• Switch selectable MPU restart address
• 200 ms Power On Reset delay
• 3 DMA modes (HALT MPU, CYCLE STEAL & MUX)
• TTY PROTO Monitor System resident in ROM
• RS1 ROM Subroutine Library resident in ROM
• ROM Resident Assembler — option
• Up to 40 ma @ 0.4V external bus loading
• 8T97 three state MPU bus drivers
• All MPU signal lines isolated & buffered
• System expansion via two 86 pin connectors

TYPICAL EVK MICROCOMPUTER APPLICATIONS
The EVK Microcomputer board allows the hardware development engineer, the logic designer, the programmer, the systems engineer, the mathematician, the scientist, the chemist or the hobbyist to have a complete working Microcomputer system, including development software by adding a low cost power supply and an ASR 33 TTY to the EVK Microcomputer board. The EVK series of Microcomputers boards allows the owner/user to use one of these boards to:
• Evaluate the complete set of AMI's family of Microcomputer IC's at a low investment of time & money — no design time is required.
• Serve as a general purpose Microcomputer for low volume systems to which the systems engineer can easily add additional I/O ports and memory.
• Serve as a low cost quick turn around prototype system to evaluate total system mechanization concept (hardware & software) and market acceptance prior to committing to a custom design system for large volume production.
• Serve as a low cost minimal 6800 Microcomputer application software development system.
• Serve as a low cost general purpose Microcomputer to run numerous application software programs.
EVK MICROCOMPUTER FUNCTIONAL DESCRIPTION

FUNCTIONAL CONFIGURATION

The following functional description is directed towards the fully populated EVK 300 Microcomputer board and is functionally applicable to the complete series of EVK boards limited only by the degree of board hardware-software population.

No attempt will be made to functionally describe the characteristics of AMI's Microcomputer IC chip set as this was undertaken in the first article entitled AMI 6800 Microcomputer Chip Set published last month in INTERFACE AGE. Instead, we will describe the general architecture of the EVK Microcomputer board and its general characteristics in order to provide an insight into EVK board utilization.

GENERAL ORGANIZATION

The EVK Microcomputer functional configuration is composed of the following major functional sections: MPU, clock, internal timer, memory, EPROM programmer, internal bus, expansion bus, I/O bus, I/O, and control logic sections. This functional inter-relationship is shown in Fig. 1. EVK Microcomputer Functional Block Diagram while the detailed logic and circuit information is shown in Fig. 2, 3, 4, & 5. EVK Microcomputer Logic Diagrams.

MPU — The MPU is mechanized with AMI's S6800 Microprocessor chip. All MPU data address and control lines are buffered, and in addition are available at the board edge connector.

MPU TWO-PHASE CLOCK — The basic MPU two phase clock is derived from a 96S02 dual one-shot (IC12) connected either in a regenerative feedback loop or driven by a 1 MHz crystal controlled oscillator circuit (IC14). Switch #SW 2 is used to select either the one-shot regenerative feedback or the crystal oscillator mode of operation. Phase one and two timing is controlled by potentiometers connected to the one-shot RC timing networks and controls the phase pulse widths. These two phase additive pulse widths determine the MPU clock rate when the one-shot regenerative feedback configuration mode is connected. In this regenerative feedback mode, MPU clock frequency may be adjusted from 300 KHz up to 1 MHz. The phase timing outputs of the one-shots in both modes of operation drive 2N5771-2N5772 transistor amplifier circuits which in turn drive the two phase clocks of the MPU. In addition, both clock phases are buffered and available at the board edge connector. The fixed frequency 1 MHz crystal oscillator circuit output is also buffered and available at the board edge connector.

The two phase clock can be halted in either phase 1 or phase 2 for cycle-steal, DMA or slow memory applications. Phase 1 is halted (held HIGH) by driving the CYCLE STEAL control line LOW. Phase 2 is halted (held HIGH) by driving the MEMORY READY line low. Because the S6800 internal registers are dynamic and must be refreshed periodically CYCLE STEAL and MEMORY READY line outputs to the one-shots cannot be held LOW for more than 5 µs. This time limit protection, regardless of control input conditions, is provided by open collector 7407 (IC65) Hex non-inverting drivers, disconnect diodes and one-shot RC pull-up timing networks.

INTERNAL TIMER — A crystal controlled interval timer provides 100µs and 1 ms time periods for interrupting the MPU for real time clock applications. The 1 MHz crystal clock output drives a three decade divide-by-ten 74160 counters (IC50, 51, & 52) which in turn provide the 100µs and 1 ms time intervals. The 100 µs time interval pulse sets bit 7 of I/O address FBC7 via the S6820 PIA (IC47) while the 1 ms time interval pulse sets bit 7 of I/O address FBC5 via the S6820 PSA. These two time interval signals are used for timing EPROM programming.

MICROCOMPUTER BUS ARCHITECTURE — The EVK Microcomputer in essence has three sets of busses, namely the MPU bus, the Microcomputer bus and the on board memory-I/O bus. Each bus set consists of an 8-bit bidirectional data bus, a 16-bit unidirectional address bus and a control bus. The MPU bus is isolated from the Microcomputer bus in order to keep MPU signal loading to a minimum. The on board memory-I/O bus is isolated from the Microcomputer bus in order to assure that the on board memory and I/O devices do not load down the Microcomputer bus. As a result of this load isolation 40 ma drive current is available to drive external expansion hardware. The bus isolation buffers are non-inverting 3-state hex buffers (8T97). All of the controls to and from the S6800 are available at the board edge connector. This allows the user-complete access and control of the MPU. Bus logic polarity is the same on all three busses (logic true = voltage high = "1"). The enable control signals to the MPU are always active. Control signals for the address bus are gated by the DMA GRANT line. The data bus is controlled by the DMA and R/W Lines.

MEMORY — The on board memory includes 1 K bytes static RAM, 4K bytes ROM and 2K bytes EPROM.

MEMORY ADDRESS ASSIGNMENTS — Address assignments have been made such that all components on the card can run in the upper 8K bytes of memory. An address assignment map is shown in Figure 6.

Address decoding is made by use of three 74S138 one-of-eight decoders (IC 44, 45, 54). The first decoder (IC 54) selects one 1K-byte block of the upper eight 8K-bytes of memory. The output of this decoder is for RAM, I/O, ROM, or PROM enable lines. The second decoder (IC 44) selects one of eight RAM memory chips. The third (IC 45) selects I/O devices on the board.

A MEMORY DISABLE line is available at the Bus edge connector. This line, when LOW, deselects the first address decoder disabling all I/O and memory devices on the board. An I/O ENABLE line is derived from the first adress decoder and is available at the Bus edge connector. It must be noted that I/O ENABLE on the backplane is not valid when MEMORY DISABLE is LOW.

READ ONLY MEMORY — The Prototyping Board has assigned locations for two 1K byte S6830 ROMs and for four 512 x 8 S6834 EPROMs. The ROM circuits are designed such that the locations will also accept two 2K byte 16K ROMs (S6831). Thus, maximum memory allocation for ROM and EPROM is 6K bytes. The prototyping operating system program (PROTO) is assigned to the ROM with a starting address of F000.
The four EPROM locations may contain any user program. Execution can start from beginning EPROM location either by selecting EPROM starting address of E000 in the restart switches or by branching to that address using the "G" command in the PROTO program.

**RANDOM ACCESS MEMORY** — The RAM is divided into two parts, 512 bytes fixed in the highest memory locations and 512 bytes of moveable memory. Since the highest memory locations (FFFF, FFFF) are used for restart address, the address circuits disable the RAM using a memory disable line and force the 16 bit switch address on the data bus whenever a Reset occurs. This allows the user to vector to any address as his restart address.

The PROTO program assigns restart vectors for IRQ, NMI, and SWI whenever it is started (usually via Reset). It is therefore important to note that the user program must do the same thing if he does not use PROTO and restarts from a power down mode.

The stack pointer is assigned to address FFBF in PROTO. This allows the remaining RAM to be used as stack if so desired.

A switch option allows 512 bytes of RAM to be relocatable. When in the upper portion of memory, the 512 bytes are addressed whenever A9 and A15 are not true (0000 — 01FF for example). It is thus recommended that RAM be assigned to the low address only if the user does not add other RAM to his development system.

**I/O** — On board I/O includes parallel PIA I/O ports and serial ACIA TTY and RS232C I/O ports.

**PARALLEL I/O** — Three S6820 PIA’s give the user a wide range of I/O flexibility. The PIA’s are assigned addresses as shown in Table 2. Interface pins of these devices are directly connected to the I/O edge connector. The CA2 pin for the PIA at addresses FBC4 is also connected to the Vprog input (pin 11) to the EPROM socket (IC 46) through a +5V to -50V driver. The user is cautioned to use this line such that it will not interfere with his I/O function if programming an EPROM. For example, if the CA2 line is connected to an external control function, this function may be erroneously activated while programming an EPROM.

**SERIAL I/O** — One S6850 ACIA allows the system to communicate bi-directionally with serial data I/O peripherals such as a TTY. A baud rate generator generates all standard communication frequencies by switch selection. This frequency operates independently of the system clock so the MPU frequency can be changed without altering the I/O clock rate. See Table 3 for switch setting and associated frequencies.

A 20 mA current loop interface and an RS-232 interface are both available at the I/O edge connector.

Address assignments for the ACIA are given in Table 3, "Bit Rate Generator Switch Settings."

**EPROM PROGRAMMER** — A unique feature of the Prototyping Board is its ability to program AMI S6834 EPROMs. A third PIA latches the address and data information for programming the EPROM. The EPROM socket programs only the S6834 EPROM, however, an adapter plug is available to also program the AMI S5204A EPROM. Except for the Vprog input, all address, chip select, R/W and data I/O pins on both EPROMs are completely TTL compatible and are driven directly from the PIA outputs. The outputs are also available on the I/O edge connector for convenience in using another EPROM programming socket.

---

**FIGURE 6. MEMORY ASSIGNMENT MAP FOR THE AMI PROTOTYPING BOARD**

---

**TABLE 2. I/O ADDRESS ASSIGNMENT**

<table>
<thead>
<tr>
<th>I/O PORT</th>
<th>ADDRESS</th>
<th>ASSIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>S6850 ACIA</td>
<td>FBC4</td>
<td>Peripheral Register A</td>
</tr>
<tr>
<td></td>
<td>FBC5</td>
<td>Peripheral Register B</td>
</tr>
<tr>
<td></td>
<td>FBC6</td>
<td>Peripheral Register C</td>
</tr>
<tr>
<td></td>
<td>FBC7</td>
<td>Peripheral Register D</td>
</tr>
<tr>
<td>S6820 PIA 1</td>
<td>FBC8</td>
<td>Control Register A</td>
</tr>
<tr>
<td></td>
<td>FBC9</td>
<td>Control Register B</td>
</tr>
<tr>
<td></td>
<td>FBC10</td>
<td>Control Register C</td>
</tr>
<tr>
<td></td>
<td>FBC11</td>
<td>Control Register D</td>
</tr>
<tr>
<td>S6820 PIA 2</td>
<td>FBC2</td>
<td>Peripheral Register A</td>
</tr>
<tr>
<td></td>
<td>FBC3</td>
<td>Peripheral Register B</td>
</tr>
<tr>
<td></td>
<td>FBC4</td>
<td>Peripheral Register C</td>
</tr>
<tr>
<td></td>
<td>FBC5</td>
<td>Peripheral Register D</td>
</tr>
<tr>
<td>S6830 PIA 3</td>
<td>FBC6</td>
<td>PROM Burner</td>
</tr>
<tr>
<td></td>
<td>FBC7</td>
<td>Control Register A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control Register B</td>
</tr>
</tbody>
</table>

---

**TABLE 2. I/O ADDRESS ASSIGNMENT**

<table>
<thead>
<tr>
<th>I/O PORT</th>
<th>ADDRESS</th>
<th>ASSIGNMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>S6850 ACIA</td>
<td>FBC4</td>
<td>Peripheral Register A</td>
</tr>
<tr>
<td></td>
<td>FBC5</td>
<td>Peripheral Register B</td>
</tr>
<tr>
<td></td>
<td>FBC6</td>
<td>Peripheral Register C</td>
</tr>
<tr>
<td></td>
<td>FBC7</td>
<td>Peripheral Register D</td>
</tr>
<tr>
<td>S6820 PIA 1</td>
<td>FBC8</td>
<td>Control Register A</td>
</tr>
<tr>
<td></td>
<td>FBC9</td>
<td>Control Register B</td>
</tr>
<tr>
<td></td>
<td>FBC10</td>
<td>Control Register C</td>
</tr>
<tr>
<td></td>
<td>FBC11</td>
<td>Control Register D</td>
</tr>
<tr>
<td>S6820 PIA 2</td>
<td>FBC2</td>
<td>Peripheral Register A</td>
</tr>
<tr>
<td></td>
<td>FBC3</td>
<td>Peripheral Register B</td>
</tr>
<tr>
<td></td>
<td>FBC4</td>
<td>Peripheral Register C</td>
</tr>
<tr>
<td></td>
<td>FBC5</td>
<td>Peripheral Register D</td>
</tr>
<tr>
<td>S6830 PIA 3</td>
<td>FBC6</td>
<td>PROM Burner</td>
</tr>
<tr>
<td></td>
<td>FBC7</td>
<td>Control Register A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Control Register B</td>
</tr>
</tbody>
</table>
TABLE 3. BIT RATE GENERATOR
SWITCH SETTINGS,
0 = CLOSED, 1 = OPEN

<table>
<thead>
<tr>
<th>SW POSITION</th>
<th>BIT RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 3 2 1</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>19,200 baud</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 baud</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>50 baud</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>75 baud</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>134.5 baud</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>200 baud</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>600 baud</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>2,400 baud</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>9,600 baud</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>4,800 baud</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1,800 baud</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1,200 baud</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>2,400 baud</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>300 baud</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>150 baud</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>110 baud</td>
</tr>
</tbody>
</table>

Programming is achieved by pulsing the VPROG pin with -50 volts through the CA2 line of the PIA at address FBC4. This line drives the transistor that gates the -50 volt source to the VPROG pin. The -50 volt source is switched ON or OFF via the VPROG switch.

CONTROL — The Microcomputer control section includes system reset logic, addressable reset logic and DMA control logic. In addition, an external logic circuit may be added to provide selection between RUN and single step modes.

RESET — The Reset circuit provides a timed reset for Power On Reset timing and for the Reset switch. The circuit is a timed oscillator which provides a 200 ms reset pulse.

RESTART — The starting address of an S6800 is FFFE/FFFF. The contents of these memory locations are put into the Program Counter register each time the MPU is reset. The Evaluation Board traps the FFE/FFFF addresses and puts the contents of the two 8-bit switch sets (IC 32, 43) on the data bus for each address and disabling memory, then gating the first set of switches to the Data Bus during FFFE time and the second set during FFFF time. The user is thus allowed to select any restart address by simply selecting a two byte address on the 16 bits of switch settings. The two DIP switches may be replaced with four hex thumbwheel switches mounted on a front panel and interconnected via a flat ribbon cable and DIP plug connectors providing front panel Hex restart control.

DMA — Three types of DAM implementation are possible on the Prototyping Board, a halt processor mode, a cycle steal mode and a multiplex mode. A switch selects these DMA modes. The switch must be in the DMA position for the multiplex DMA mode. A delayed clock gives the DMA GRANT line to the bus after the "Data Hold" time has passed for a multiplexed type of DMA operation. The control lines for the halt processor and cycle steal modes are available at the Bus edge connector.

RUN/HALT & SINGLE STEP EMBELLISHMENTS — A simple low cost three IC RUN/HALT-Single Step Instruction logic may be added external to the EVK board to provide these capabilities if required. Figure 7, RUN/HALT & Single Cycle Instruction Logic Diagram and Figure 8, Single Step Timing Diagram depicts this added logic mode.

POWER REQUIREMENTS

The EVK board is mechanized so that nominally only a +5 volt @ 3.5 amp power supply is required. A -12 volt supply is required when using S6834 EPROM ICs. In addition, a -50 supply is required when programming these EPROMs. The RS232C Interface requires both the +12V and -12V supplies for proper operation. The following is the total power and voltage level requirement for a complete operational EVK 300 Microcomputer board:

+5V @ 4 Amps
-12V @ 150 ma
+12V @ 50 ma
-50V @ 50 ma

SOFTWARE

The EVK 300 Prototyping Board Software is comprised of a TTY Operating Program (PROTO) and is supported by a ROM Subroutine Library (RS). PROTO — The EVK 300 is supplied with a prototyping operating system program (PROTO). The program resides in ROM with a starting address of F000. The various routines within PROTO are called by entering via the TTY keyboard one of the commands. A command consists of one character command identifier followed by additional parameters, if needed, separated by blanks or commas. All commands end with a carriage return. Since no action is taken before the carriage return, an input line may be deleted by the use of the TTY ESCAPE key. The PROTO program operates on the following commands:

L Load Memory from TTY paper tape (HEX Format)
P Punch a Memory location to TTY paper tape (HEX Format)
S Set (write) Memory to a given value
D Display the contents of a memory location in HEX
G Go to user program at specific address and begin program execution
R Print contents of MPU C, B, A, X, P & S register on the TTY
B Burn (program) an EPROM from Memory location indicated
V Verify the contents of an EPROM with a specified memory location
I Input (copies) contents of EPROM in the programming socket into memory
M Move a specific block of memory to a designated location
E End of transmission (EOT) character terminates the record and punches EOT on paper tape.

The commands will operate on a single character op code plus address parameters from the TTY keyboard.
(RS)³ SUBROUTINES — The 2K X 8 ROM provided with the PROTO prototyping system includes a set of (RS)³ subroutines with a slightly different linkage from the standard (RS)³ form, although the calling sequence is the same. In particular, the provision for additional subroutines in the of other (RS)³ ROMs is limited to a total of 127 subroutines. The first additional (RS)³ ROM address must be placed in RAM location FFF4 (which can be set via the Set Memory command or modified by an initialization code in a user program). Also, since it is incorporated into a larger program, the whole of which very nearly fills the 2K bytes of its ROM, the (RS)³ part of the ROM does not start on an even page boundary, making it awkward for isolated use. However, the 24 subroutines included in this ROM are available to user program calls with the SWI calling sequence, as described.

The ROM Subroutine Library (RS)³ operates on a single SWI (3F) command and a second byte of offset giving the S6800 an additional set of two-byte instructions. Specific subroutines (offsets) are as follows.

Figure 7. Run/Halt and Single Cycle Instruction Logic Diagram

Figure 8. Single Step Timing Diagram
FUNCTION

0  PUSHALL  All registers are pushed on to user stack.
1  POPALL   All registers on user stack are loaded into MPU.
2  TXAB     Contents of Index Register are transferred to A & B Accumulators.
3  TABX     Contents of A & B Accumulators are transferred to Index Register.
4  XABX     Contents of A & B Accumulators are exchanged with contents of Index Register.
5  PUSX     Contents of Index Register are pushed onto user stack.
6  PULX     Index Register is loaded with contents of user stack.
7  ADDXAB   Contents of Index Register are added to contents of A & B Accumulators. Sum is in A & B Accumulators.
8  ADDABX   Contents of A & B Accumulators are added to contents of Index Register. Sum is in Index Register.
9  ADDAX    Contents of Accumulator A are added to contents of Index Register. Sum is in Index Register.
10 ADDBX    Contents of Accumulator A are added to contents of Index Register. Remainder is in Index Register.
11 SUBXAB   Contents of Index Register are subtracted from contents of A & B Accumulators. Remainder is in Accumulators A & B.
12 SUBABX   Contents of Accumulators are subtracted from contents of Index Register. Remainder is in Index Register.
13 SUBAX    Contents of Accumulator A are subtracted from contents of Index Register. Remainder is in Index Register.
14 SUBBX    Contents of Accumulator B are subtracted from contents of Index Register. Remainder is in Index Register.
15 P2HEX    Two Hexidecimal Characters (one MPU byte) are printed on the TTY.
16 P4HEX    Four Hexidecimal Characters (two MPU bytes) are printed on the TTY.
17 PRINTA   ASCII Character designated is printed on TTY.
18 PMESS    Message designated is printed on TTY.
19 VALAN    Character (byte) is checked to see if it is a valid alpha/numeric character.
20 INPUTA   ASCII Character at TTY is input to MPU.
21 CONHB    ASCII Character string is scanned looking for a valid Hexidecimal character. Binary equivalent is returned in Accumulators A & B.
22 INDEX    Contents of Accumulator A are multiplied with the contents of Accumulator B and the product is added to the contents of the Index Register.
23 MUL8     Contents of Accumulator A are multiplied with the contents of Accumulator B. Product remains in both Accumulators.

S6800 MICRO ASSEMBLER/DISASSEMBLER (MA/D) —
An optional ROM resident Micro Assembler/Disassembler is available for the EVK Microcomputer board at an additional cost of $30.00. Where this option is provided for those applications it may be desirable to debug programs using the mnemonic instruction codes instead of hexadecimal values. MA/D is designed to accomplish this by interfacing with a user via a keyboard and display (TTY or equivalent). The required 6800 environment must include:

Character in routine at location 0
Character out routine at location 3
No. nulls after carriage return at location 6
RAM at locations 7-78 10

The I/O routines must transfer the characters in Register A and return with a RTS. It is expected that location 0 will just include a JMP to the actual character in routine, or, in the case of AMI's proto board:

00 SWI
01 FCB 20
02 RTS

The stack pointer must also be initialized before MA/D is entered. MA/D itself can execute from ROM, located anywhere in the system. MA/D may be started at its beginning address +2, in which case it will set up its environment for the AMI proto board.

After entering MA/D, the line length may be changed. The line length is in location 7 and is initially set to (20)10 = 14 hex. The line buffer itself begins in location (58)10 = 3A hex.

After displaying a header message MA/D prompts the user for a command by displaying MA/D's current location counter followed by a colon (:). The commands available to the user allow for disassembly of instructions in memory and assembly (mnemonic translation and operand insertion with relative offset computation) of instructions directly into memory.

MA/D is also very useful for writing short test programs. The instruction format for assembly is identical to the S6800 Assembler except:

1) operands must be in hexadecimal without the $, and no more than four digits long
2) no symbols can be defined or referenced
3) relative addresses are specified as absolute addresses, the offset is computed
4) in those instructions having both direct and extended addressing modes, extended addresses must have at least three digits. Thus, LDA A 10 assembles as 96 10
   LDA A 010 assembles as B6 00 10
5) in those instructions not having a direct addressing mode, the operand may be two or more digits. Thus,
   INC 10 assembles as 7C 00 10
6) an operand may be a single hex digit only if the op code indicates an A or B register, or immediate mode addressing. Thus,
   INC 01  INC 1
   LDA A 1  LDX 1
   LDX #1
   —are legal—    —are not—
   (This makes it easier to distinguish between, for instance, INC A and INC OA.)
7) Anywhere a number is used, the construction 'character may be used instead, and is equivalent to the ASCII code for the character.
MA/D Command Summary

@newloc
The @ sign followed (immediately or with blank separator) by a hexadecimal address initializes the current location counter to the new address. MA/D automatically updates the location counter as instructions are assembled or disassembled.

$count
The $ sign followed by a one or two digit (hexadecimal) count results in the disassembly of “count” instructions. Zero = infinity.

!address
The exclamation mark followed by an address causes MA/D to call a subroutine at the given address. If the subroutine returns with the carry flag set, MA/D will print “????”.

! Exclamation mark with no address given causes MA/D to call the subroutine starting at the current location.

“string
Assembles the ASCII characters following the double quote mark into successive bytes of memory starting at the current location. The current location is updated.

xx
A one or two digit hexadecimal number is placed into the current location, and the current location is incremented by one. A single quote mark followed by a single character causes the ASCII code for that character to be placed in the current location.

This command may appear several times on the same line, the numbers or quoted characters separated by spaces or commas.

The commands to MA/D are buffered and not processed until the <RETURN> key is depressed. The <BACKSPACE> key can be used to delete the last character input. If errors are detected on user input the line is ignored, ??? is displayed, and another prompt is issued.

The default command is “assemble” and MA/D, if not recognizing the input as one of the following commands, generates the machine code for the instruction mnemonic.

Next month we will publish the complete PROTO Assembly Listing for the EVK Microcomputer board.

The commands to MA/D are buffered and not processed until the <RETURN> key is depressed. The <BACKSPACE> key can be used to delete the last character input. If errors are detected on user input the line is ignored, ??? is displayed, and another prompt is issued.

The default command is “assemble” and MA/D, if not recognizing the input as one of the following commands, generates the machine code for the instruction mnemonic.

Next month we will publish the complete PROTO Assembly Listing for the EVK Microcomputer board.

The commands to MA/D are buffered and not processed until the <RETURN> key is depressed. The <BACKSPACE> key can be used to delete the last character input. If errors are detected on user input the line is ignored, ??? is displayed, and another prompt is issued.

The default command is “assemble” and MA/D, if not recognizing the input as one of the following commands, generates the machine code for the instruction mnemonic.

Next month we will publish the complete PROTO Assembly Listing for the EVK Microcomputer board.
INTRODUCTION

This article is Part Three of a series on the EVK Microcomputer hardware, firmware and supporting software. This month's subject covers the ROM resident Prototyping TTY MONITOR Operating System, PROTO.

PROTO SOFTWARE

The resident PROTO software program includes the following commands:

- **L** LOAD HEX paper tape program into RAM memory
- **P** PUNCH HEX paper tape from memory
- **S** SET (write) specified data string characters into consecutive memory locations
- **D** DISPLAY (prints) in HEX to TTY contents of specified memory locations
- **G** GO TO user program at specified address and execute
- **R** PRINTS contents of MPU register (C, B, A, X, P & S) on TTY at time the user's program was last interrupted
- **B** BURN (copies) the contents of specified memory into the EPROM in the programming socket
- **V** VERIFY (compares) contents of specified memory with EPROM or ROM in the programming socket
- **I** INPUT (copies) contents of the EPROM or ROM in the programming socket into specified RAM memory locations
- **M** MOVE (copies) contents of memory block from specified location to designated RAM memory location
- **E** END of transmission (EOT) character terminates the end of punch paper tape record and punches EOT on paper tape.

The commands will operate on a single character OP CODE plus address parameters from the TTY keyboard.

PROTO COMMAND DESCRIPTIONS

The EVK 300 board will be supplied with a prototyping operating system program (PROTO). The program resides in ROM with a starting address of F000. The various routines within PROTO are called by entering via the TTY keyboard one of the commands described in the following paragraphs. A command consists of one character command identifier followed by additional parameters, if needed, separated by blanks or commas. All commands end with a carriage
return. Since no action is taken before the carriage return, an input line may be deleted by the use of the TTY ESCAPE key.

L, ADDL, ADDH, OFFSET

The Load tape command loads data from a hex formatted tape (see paragraph on 6800 HEX tape format at end of article) into the user's memory between ADDL and ADDH, inclusive. The OFFSET is added to the memory address specified on the tape to form the actual memory starting address for the data stored. If a byte to be stored into memory has an address outside of the range ADDL, ADDH, it is not entered into memory, but a Delete character (H'FF) is transmitted to the terminal.

Example: L 0100 02FF FFFA

The address range in the L command is optional, and if omitted is assumed to be the full range of memory (0000-FFFF). The offset parameter is also optional, and if omitted is assumed to be zero (0000). Thus the L command with no parameters loads the tape into the memory locations specified on the tape with no offset. The offset value in the L command is a two's complement signed number, entered in unsigned hexadecimal. For example, an offset of -6 is entered as FFFA.

If an attempt is made to load non-existent memory, or ROM, the loading operation will terminate, typing out the address and the message "BAD ADR."

In operating the Load command, PROTO turns on the tape reader and scans the tape for the first ASCII "S," which indicates start of record. It is not necessary to position the tape at the first record of a tape file since each record contains its own starting address. PROTO will load data records until it encounters an end of file (EOF) record or a tape error (Check Sum or illegal character). When PROTO reads a header record (start of record and address), it translates the header into ASCII characters and prints the result. The Check Sum is the binary sum of all characters in the block. PROTO does not list the tape contents as the tape is being read.

When PROTO encounters an end of file record or a tape error, it turns off the reader and prints "EOF" or "CKSM ERR" respectively.

P, ADDL, ADDH, OFFSET

The Punch hex format command causes PROTO to punch on the TTY paper tape the contents of memory between ADDL and ADDH, inclusive. Each record is punched with a four-digit hex address of the starting byte of the record. This address is derived from the memory address of the byte being punched, plus the offset value, OFFSET. The offset is optional, and if omitted is assumed to be zero.

All data records are punched in hex format. Records using this command (except the last record) contain 16 bytes of data plus the start code, byte count, address, and the checksum.

The P command does not cause an EOF record to be punched so that several disjoint blocks of memory can be combined on one tape file.

Example: P F000 F07F 0F00

S, ADDR, BYTE1, BYTE2, ..., BYTEN

The Set memory command writes the 8-bit data words specified by BYTE1 to BYTEN into consecutive memory locations starting at ADD.

If ADD has more than 4 (hexadecimal) characters or if any of the data bytes have more than 2 characters each, only the last 4 or 2 characters are used respectively.

Example: S 0000 86 05 97 28

Memory locations at 0000 through 0003 are loaded as shown.

D, ADDL, ADDH

The Display memory command prints the contents of memory between ADDL and ADDH, inclusive, in hex format. Up to sixteen bytes per line are printed, preceded by the hexadecimal address of the first byte of the line. A carriage return is forced after a byte having a low order digit of F in its memory address is printed.

Example: D FC00 FC1F

Two lines of memory contents are printed as follows:

FC00 00 01 02 03 04 ... EOF
FC10 10 11 12 13 14 ... 1E1F

G, ADDR

The Go command starts execution of the user program at the address specified by the input parameter. To insure that all registers contain the same information they held before the user program was interrupted, PROTO pushes into the stack the copy of the user registers that it keeps at locations FFEB—FFF3 (CC, B, A, X, P, S) then executes an RTI instruction. The user can change the initial values of the

Good Software and Support are to a computer as the driver is to his car. One without the other and you have a magnificent paperweight.
registers by changing the contents of these locations.

Example: G 300

Program will branch to address 0300 and start execution from that point.

R

The Registers command prints the contents of memory locations FFEF—FFFF which contain the values that were in the user's C, B, A, X, P, and S registers (in that order) when the user's program was last interrupted.

B, ADDL, ADDH, ROMAD

The Burn command copies the contents of user memory into the EPROM in the programming socket, beginning with memory location ADDL through ADDH, inclusive, to EPROM locations beginning with address ROMAD. Each byte is burned in with 20 3-ms pulses of −50V on the Vprog pin (pin 11) of the EPROM. Before attempting to write into the EPROM, the contents of the EPROM are compared with the user memory data byte to verify that the EPROM will take the byte (PROTO will not attempt to program a EPROM location to logic LOW which already contains logic HIGH). After the 20 pulses, the new contents of the EPROM are verified against the memory byte to be sure the data was indeed written. If the byte did not program, a NAK code is typed out on the terminal, and another try is made, up to a maximum of three tries.

If the preverify encounters a EPROM location containing HIGHs where the memory byte has zeros, PROTO will type out the memory address, the memory byte in binary, the EPROM byte in binary, and the EPROM address (if different from the memory address), then stop. If after attempting to write data into the EPROM, the data does not program, or erroneous bits show up, a similar display occurs for the failing location, with the additional message “BAD ADR.” typed on the same line.

The EPROM address ROMAD is optional, and if omitted, ADDL is used, with only the least significant nine bits of the address being used. If the address range ADDL, ADDH is omitted, the 512 bytes beginning at FC00 are used, and the EPROM is checked to insure it contains all LOWs before any locations are written. If not, four question marks are typed and the B command is aborted.

V, ADDL, ADDH, ROMAD

The Verify command compares user memory between ADDL and ADDH, inclusive, with the corresponding locations in the EPROM in the programming socket, beginning with EPROM address ROMAD. Each location that does not match is typed out in the following format:

aaaa mmmmmmmm ppppppppppp rrrr

where “aaaa” represents the user memory address, “mmmmmmmmm” represents the memory byte, in binary, and “rrrr” represents the EPROM address, if different from the memory address (in the low nine bits). Nothing is typed for matching locations. The typeout may be aborted by typing an ESC key during the typeout.

If the ROMAD parameter is omitted, ADDL is assumed. If no parameters are supplied in the command, the whole EPROM is compared to the contents of FC00 — FDFF.

I, ADDL, ADDH, ROMAD

The Input command copies the contents of an EPROM in the programming socket into memory beginning at the address ADDL through ADDH, inclusive, from the EPROM address ROMAD. If ROMAD is omitted, ADDL is assumed. If no parameters are supplied, the entire EPROM is copied into the RAM area, FC00 — FDFF. An attempt to copy an EPROM into non-existent memory will abort the command with the message “BAD ADR.”

M, ADDL, ADDH, DEST

The Move command copies memory from the range ADDL — ADDH, inclusive, to the RAM locations starting at DEST. This copy begins at the lower address, so if DEST lies within the range ADDL — ADDH, some of the original data will be lost, and other parts will be duplicated.

E

The End of Transmission command is used to cause an EOT character to be punched on the paper tape. After a field has been punched, an EOT will terminate the record and punch a trailer tape. When reading a record, the reader will stop at the EOT character. If no EOT character is present, the reader must be manually turned off and the Reset switch must be pressed to enter the operating system program.

THE SUBROUTINE ROM

Many of the monitor's functions are accomplished with the help of the Re-Entrant Self-Relative Subroutine ROMs (RS)3. This standard ROM, which can be considered a software extension to the 6800 instruction set, is also available to be used by the user both on the prototype board and in his final production system. The user can call one of the 25 (RS)3 subroutines with an SWI instruction followed by the number of the desired subroutine.

The user should be aware of the fact that the (RS)3 pushes from 7 to 10 bytes of data onto the stack, depending upon which subroutines are called. This means that if the user calls (RS)3 routines, he must make sure that the necessary memory space is available for stack expansion.

Since PROTO assigns its own stack area, the user need not be concerned about how (RS)3 is used.

INTERRUPTS

Of the four available interrupt vectors, IRQ, RESET and SWI are used by PROTO while NMI is left for the user. The vectors are in RAM (except for RESET which is switch controlled) so the user writing his own program can completely control the system.

The upper memory locations are RAM. If the user
SOFTWARE SECTION

expects either NMI or IRQ interrupts to occur, he must initialize the vector addresses to the starting address of the IRQ and NMI handler routines.

PROTO must have control of the RESET vector so that the RESET switch on the Prototyping Board can return program control to PROTO at any time.

The reset routine copies the contents of the B, A, X, CC, and S registers into a fixed area of memory. This means that the program can be aborted at any time by using the reset switch while still saving all the registers except the program counter. Unfortunately, the contents of the program counter are lost.

It is possible for the user to use the NMI interrupt to abort a program execution without losing the contents of the P and C registers. This condition is automatically set in the NMI handling routine when PROTO is called. This interrupt vector will cause the contents of the user's registers to be printed when the NMI line goes low.

Since the SWI instruction is used to call subroutines between 00 and H'18 from (RS)\(^3\), the user is somewhat limited in the ways he can use SWI instructions. However, he can access an SWI handler routine in his own program by an SWI instruction followed by a byte containing the decimal number less than H'80 but greater than H'19 < n < H'80 sequence. PROTO passes control at address FFF4. If the user expects to access his own SWI routine and use PROTO, he must use the Set Memory command to store the address of this routine at locations FFF4 and FFF5.

PROTO makes sure that the user's SWI routine is entered from the stack with all registers containing the same information that they would hold if the routine were entered directly through the SWI vector.

BREAKPOINTS

Breakpoints allow the user to halt his program and examine the contents of the internal registers. PROTO provides two types of breakpoints. In this system, breakpoints are actually debugging routines that can be called from the user's program just like (RS)\(^3\) routines.

Each breakpoint requires a two byte calling sequence: an SWI instruction followed by a number.

Breakpoints may be inserted either by reassembling the program with the extra SWI instructions added or the Set Memory command may be used to replace parts of the code with SWI instructions. Note that the second method is not satisfactory for the snapshot option (described below) since the replaced code must be restored before execution can be continued. When using the second method, the user must make sure that he replaces the first two bytes of an instruction. If the SWI replaces the second or third byte of an instruction, it may be interpreted as an address rather than an opcode.

The different types of breakpoints are:
1. Print registers (SWI, H'80)
2. Snapshot (SWI, H'81)

The sequence SWI, H'80 saves the user's registers at the vector stored in FFF4 — FFF5, prints their contents (in the order CC BB AA XXXX PPPP SSSS), then returns control to PROTO.

The sequence SWI, H'81 prints out the contents of the user's registers then continues executing the user's program starting at the address following the byte containing the number H'81. Note that if this address does not contain a valid opcode, unpredictable results will occur.

MICROCOMPUTER DEVELOPMENT SOFTWARE

6800 PAPER TAPE HEX FORMAT

The AMI 6800 Hex Tape format provides a compact representation of binary data patterns for transmission using ASCII communication terminals.

The Hex tape is organized into data records with each record containing information in the same format. The record information consists of type, length, address, data and checksum. All records begin with an 'S' character for start of record identification. All information on the tape which is not between a start of record and the checksum is ignored.

TAPE FORMAT

<table>
<thead>
<tr>
<th>ASCII Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start of record (S)</td>
</tr>
<tr>
<td>2</td>
<td>Type of record</td>
</tr>
<tr>
<td></td>
<td>0 — Header record</td>
</tr>
<tr>
<td></td>
<td>1 — Data record</td>
</tr>
<tr>
<td></td>
<td>9 — End of file record</td>
</tr>
<tr>
<td>3—4</td>
<td>Byte Count</td>
</tr>
</tbody>
</table>
<pre><code>                    | Since each data byte is represented as two hex characters, the byte count must be multiplied by two to get the number |
</code></pre>

FEBRUARY 1977

INTERFACE AGE 113
of characters to the end of the record. (This includes checksum and address data.)

5, 6, 7, 8

Address Value
The memory location where this record is to be stored.

9, ..., N

Data
Each data byte is represented by two hex characters.

N+1, N+2

Checksum
The one’s complement of the additive summation (without carry) of the data bytes, the address, and the byte count.

Example Data Record

Memory Contents

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A000</td>
<td>10</td>
</tr>
<tr>
<td>A001</td>
<td>1A</td>
</tr>
<tr>
<td>A002</td>
<td>20</td>
</tr>
<tr>
<td>A003</td>
<td>2A</td>
</tr>
</tbody>
</table>

Data Record Contents

<table>
<thead>
<tr>
<th>Character</th>
<th>Header Record</th>
<th>Data Record</th>
<th>End-of-File Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start of record</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>2</td>
<td>Type of record</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Byte count</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Data byte 1</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Data byte 2</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>Data byte 3</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>Data byte 4</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>17</td>
<td>Checksum</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>

The format for all hex tape records is diagrammed below.
AMI's Re-entrant self-relative Subroutine ROM: (RSRSRS) = (RS)^3

Edited by Robert A. Stevens

FOREWORD

This software article is the last of a four part series on the EVK 6800 microcomputer hardware, firmware and supporting software. This month's article covers the EVK re-entrant self-relative subroutine program library software resident in ROM.

INTRODUCTION

The cost of microprocessor software development involves many small items: the cost of assembly time, storage time, transmission time, loading time, design, development, documentation and debug. The cost of many of these items continues to accumulate even though a subroutine library exists for common functions, in particular the time and cost of transmission, loading and ROM pattern generation.

The purpose of Re-entrant Self-Relative Subroutine ROMs (RS)^3 is to give the user a hardware subroutine package which exists in the breadboard design from the beginning. The programs are documented, debugged and constitute some of the most commonly performed subroutines that assembly language programmers generate.

CONCEPTS

The (RS)^3 uses a number of concepts to allow flexibility in the user environment. The first concept is self-relative programming. This simply means that the program will function correctly regardless of where it is located in memory. The user will need to know where it is located so he can reference it. However, this actual location will only have to be recorded once. The self-relative program uses relative address instructions for program control and the index and stack pointer instructions for data manipulation.

The stack is used for temporary storage of data to prevent (RS)^3 from being tied to fixed addresses. This allows the program to be re-entrant; i.e. the program can be called at different times without completing the previous call. This means that the same routine can be called by the interrupt processor as well as by the program which was interrupted. The concept of re-entrant code is not to be confused with recursive code; even through recursive coding could have been used in the subroutine package, it is not.

The subroutine calling mechanism uses the SWI instruction followed by a single byte index for the particular subroutine invoked. This was chosen because the SWI from an internal programming viewpoint is the most convenient and the safest. It is safe because an error in a ROM can be corrected by replacing the subroutine ROM without altering any other user ROM. If direct addresses to subroutine code exist in the user's domain, his ROMs would change if the location of the routine in the (RS)^3 changed.

IMPLEMENTATION

The user places the base address of the (RS)^3 into the SWI vector address. Each SWI instruction requires an index byte to follow the SWI instruction where the index indicates the function to be executed. After the function is performed, the user program will continue with the instruction following the index byte. In essence, a whole new set of instructions have been created for the user which are two bytes long.

To make the entry easier, a macro call can be provided which will assemble the correct index byte when the function name is used. A set of EQU assembler commands associates the name and the index byte value.

Example:

<table>
<thead>
<tr>
<th>MUL8</th>
<th>EQU</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL16</td>
<td>EQU</td>
<td>11</td>
</tr>
<tr>
<td>DIV8</td>
<td>EQU</td>
<td>12</td>
</tr>
<tr>
<td>DIV16</td>
<td>EQU</td>
<td>13</td>
</tr>
<tr>
<td>FUN</td>
<td>MACRO</td>
<td>INDEX</td>
</tr>
<tr>
<td>SWI</td>
<td>BYTE</td>
<td>INDEX</td>
</tr>
<tr>
<td>MEND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUN</td>
<td>MACRO</td>
<td>MUL8</td>
</tr>
</tbody>
</table>
Each (RS)³ ROM will have the ability to interrogate the index byte and vector to the appropriate subroutine if it is included in the ROM. If the index extends the number of subroutines included on the ROM, the number is subtracted from the temporary index value and the next (RS)³ ROM is automatically branched to. This allows the user to select any of several subroutine sets, where each set of subroutines is represented by a separate ROM. The selected ROMs are concatenated together into a contiguous region of the user's memory space, and are automatically linked together by the index value. Thus the actual value of the index byte for any particular subroutine is the sum of the total number of subroutines in the physically previous (RS)³ ROMs plus the offset in its own ROM. It must be noted that address assignments for (RS)³ ROMs must be made beginning at 1 K boundary addresses.

The 2K X 8 ROM provided with the PROTO prototyping system includes a set of (RS)³ subroutines with a slightly different linkage from the standard (RS)³ form, although the calling sequence is the same. In particular, the provision for additional subroutines in the form of other (RS)³ ROMs is limited to a total of 127 subroutines. The first additional (RS)³ ROM address must be placed in RAM location FFF4 (which can be set via the Set Memory command or modified by an initialization code in a user program). Also, since it is incorporated into a larger program, the whole of which nearly fills the 2K bytes of the ROM, the (RS)³ part of the ROM does not start on an even page boundary, making it awkward for isolated use. However, the 24 subroutines included in this ROM are available to user program calls with the SWI calling sequence, as described.

(RS)³ SUBROUTINES

The ROM Subroutine Library (RS)³ operates on a single SWI (3F) command and a second byte of offset giving the S6800 an additional set of two-byte instructions.

Each of the subroutines in the ROM are described, giving the index for the call, a mnemonic subroutine name, and a descriptive title. A brief description of the subroutine operation is also given.

<table>
<thead>
<tr>
<th>SUBROUTINE INDEX NUMBER</th>
<th>MNEMONIC</th>
<th>FUNCTIONAL TITLE AND DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 PUSHALL</td>
<td>Push All Registers</td>
<td>Five bytes are pushed onto the stack, containing, respectively, the Condition Codes, the B and A accumulators, and the Index Register. No registers are altered (except the Stack Pointer, which is decremented by 5).</td>
</tr>
<tr>
<td>01 POPALL</td>
<td>Pop (= Pull) All Registers</td>
<td>Five bytes are pulled from the stack into the Condition Codes, the B and A accumulators, and the Index Register, respectively. The Stack Pointer is incremented by 5.</td>
</tr>
<tr>
<td>02 TXAB</td>
<td>Transfer Index Register to A and B</td>
<td>The most significant eight bits of the Index Register are copied to the A accumulator, and the least significant eight bits are copied to the B accumulator.</td>
</tr>
<tr>
<td>03 TABX</td>
<td>Transfer A and B to Index</td>
<td>Accumulator A is copied to the most significant byte position of the Index Register, and accumulator B is copied to the least significant byte position of the Index Register.</td>
</tr>
<tr>
<td>04 XABX</td>
<td>Exchange A and B with Index</td>
<td>The contents of the Index Register and the two accumulators are exchanged, A with the most significant byte of X, B with the least significant byte.</td>
</tr>
<tr>
<td>05 PUSHX</td>
<td>Push Index Register</td>
<td>The contents of the Index Register is pushed onto the stack. The Stack Pointer is decremented by two.</td>
</tr>
<tr>
<td>06 PULLX</td>
<td>Pop (= Pull) Index Register from stack</td>
<td>Two bytes are pulled from the stack into the Index Register, and the Stack Pointer is incremented by two.</td>
</tr>
<tr>
<td>07 ADDXAB</td>
<td>Add Index to A and B</td>
<td>Add the contents of the Index Register to the two accumulators, as a 16-bit sum, leaving the result in the two accumulators. The most significant byte is assumed to be in accumulator A. The condition codes are set according to the result.</td>
</tr>
</tbody>
</table>

Condition Codes:
- H = carry from bit 11 to bit 12 of sum
- N = bit 15 of sum
- Z = 1 if sum is zero; else = 0
### SOFTWARE SECTION

**V** = 1 if two's complement overflow

**C** = carry out of bit 15 of sum

**OA**

08 **ADDABX** Add A and B to Index Register

Add the contents of the two accumulators to the Index Register, leaving the 16-bit sum in the Index Register. Accumulator A is assumed to be more significant than accumulator B. The condition codes are set according to the result.

Condition Codes:  
- **H** = carry from bit 11 to bit 12 of sum
- **N** = bit 15 of sum
- **Z** = 1 if sum is zero, = 0 otherwise
- **V** = 1 if two's complement overflow
- **C** = carry out of bit 15 of sum

09 **ADDAX** Add A to Index Register

Add the A accumulator to the contents of the Index Register, and return the sum to the Index Register. The Condition Codes are set according to the result.

Condition Codes: (Same as ADDABX)

0A **ADDBX** Add B to Index Register

Add the contents of the B accumulator to the Index Register, and leave the sum in the Index Register. The Condition Codes are set according to the result.

Condition Codes: (Same as ADDABX)

0B **SUBXAB** Subtract Index from A, B

Subtract the contents of the Index Register from accumulators A and B as a 16-bit difference. The Condition Codes are set according to the result.

Condition Codes:  
- **H** = undefined
- **N** = bit 14 of difference
- **Z** = 1 if result is zero, = 0 otherwise
- **V** = 1 if two's complement overflow
- **C** = borrow into bit 15 of difference

0C **SUBABX** Subtract A and B from Index Register

Subtract the contents of the A and B accumulators from the Index Register, leaving the difference in the Index. The Condition Codes are set according to the result.

Condition Codes: (Same as SUBXAB)

0D **SUBAX** Subtract A from Index Register

Subtract the contents of the A accumulator from the contents of the Index Register and return the difference to the Index Register. The Condition Codes are set according to the result.

Condition Codes: (Same as SUBXAB)

<table>
<thead>
<tr>
<th>Code</th>
<th>Microcomputer Development Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>0E</td>
<td><strong>SUBBX</strong> Subtract B from Index Register</td>
</tr>
<tr>
<td></td>
<td>Subtract the contents of the B accumulator from the Index Register, leaving the difference in the Index Register. The Condition Codes are set according to the result.</td>
</tr>
<tr>
<td></td>
<td>Condition Codes: (Same as SUBXAB)</td>
</tr>
<tr>
<td>0F</td>
<td><strong>P2HEX</strong> Print Byte in Hex</td>
</tr>
<tr>
<td></td>
<td>The byte pointed to be the address in the Index Register is converted to hexadecimal notation in ASCII, and output to the ACIA located as follows: Memory locations FFF6—FFF7 contain an address of a pair of bytes (indirect pointer) which in turn contain the address of the ACIA Status register.</td>
</tr>
<tr>
<td></td>
<td>FFF7</td>
</tr>
<tr>
<td></td>
<td>FFF6</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>a + 1</td>
</tr>
<tr>
<td></td>
<td>ACIA Data</td>
</tr>
<tr>
<td></td>
<td>ACIA Status</td>
</tr>
<tr>
<td></td>
<td>Each byte of the output is stored into the ACIA Data Register after bit 1 of the Status Register is true. The Control Register of the ACIA is not altered, and the Data Register is not read by this routine. The Index Register is incremented past the byte which is output.</td>
</tr>
<tr>
<td>10</td>
<td><strong>P4HEX</strong> Print Address in Hex</td>
</tr>
<tr>
<td></td>
<td>The two bytes in memory pointed to by the Index Register are converted to four ASCII digits and output to the ACIA located at the address pointed to by the pointer pointed to by the byte pair at FFF6—FFF7 (see P2HEX). The Index Register is incremented by two.</td>
</tr>
<tr>
<td>11</td>
<td><strong>PRINTA</strong> Print the Byte in A</td>
</tr>
<tr>
<td></td>
<td>The byte in accumulator A is output to the ACIA, the address of whose address is the locations FFF6—FFF7. No registers are altered except the ACIA Data Register.</td>
</tr>
<tr>
<td>12</td>
<td><strong>PMSG</strong> Print Message String</td>
</tr>
<tr>
<td></td>
<td>A message string, the first byte of which is pointed to by the Index Register, is output to the ACIA, the address of whose address is in locations FFF6—FFF7. The string is terminated by an ASCII EXT (= hex 04), and the Index Register is left pointing to that byte on return.</td>
</tr>
<tr>
<td>13</td>
<td><strong>VALAN</strong> Validate AlphaNumeric</td>
</tr>
<tr>
<td></td>
<td>The character pointed to by the Index Register is analyzed, and the Carry flag is set if it is a letter or digit; if it is not a hexadecimal digit, the Overflow flag is set. Other than the condition codes, no registers are altered.</td>
</tr>
<tr>
<td>Exit:</td>
<td>Condition Codes:</td>
</tr>
<tr>
<td></td>
<td><strong>H</strong> = undefined</td>
</tr>
<tr>
<td></td>
<td><strong>N</strong> = undefined</td>
</tr>
<tr>
<td></td>
<td><strong>Z</strong> = 0</td>
</tr>
<tr>
<td></td>
<td><strong>V</strong> = 0 if character in range 0—9, A—F; else = 1</td>
</tr>
</tbody>
</table>
14 INPUTA  Input ACIA byte to A 
One byte is input from the ACIA, 
the address of whose address is at 
location FFF6—FFF7, and this byte 
is returned to accumulator A. The ACIA 
is not written to, and except for the 
A accumulator, no registers are 
changed. (RS)* samples bit 0 of the 
status register of the ACIA, and when 
it goes to one, reads the Data Register. 
The input byte has bit 7 removed 
(set to zero).

15 CONHB  Convert Hex String to Binary 
A string of characters in memory 
beginning at the address in the Index 
Register is scanned for valid Hexa­
decimal digits; when one is found, 
and it and all immediately following hex 
digits are converted to a binary number, 
which is left in the A and B accumulators 
(A is more significant). When 
this routine is called, the maximum 
length of the string is in the B register. 
On exit, the Carry flag is set to one 
if the conversion resulted in a valid binary number, 
and the Index Register is left pointing to 
the next character in the string, or 
if the string is exhausted before finding 
any hex digits, to the last character 
of the string. Max string length in 
B is (< 128).

Condition 
Codes:  H = undefined 
       N = undefined 
       Z = undefined 
       V = undefined 
      C = 1 if valid number; 
           = 0 if not

16 INDEX  Multiply A X B and Add to Index 
The contents of the A accumulator 
is multiplied by the contents of the 
B accumulator, and the product is 
added to the Index Register. The 
Condition Codes are set according to 
the Result.

Condition 
Codes:  (Same as ADDABX)

17 MUL8  Multiply A Times B 
Multiply the contents of the A 
accumulator times the content of the 
B accumulator, and leave the product 
in both accumulators as a 16-bit 
number, with the most significant 
part in A. This is an unsigned multiply, 
and if either or both of the 
factors is negative (two's complement 
signed) the product will not be a true 
signed product of the signed factors, 
as may be seen in this formula: 
(-m)(x)m = (256-m)x = 256m + (-m). 
The condition codes are nonetheless 
set according to the result.

Condition 
Codes:  H = undefined 
       N = bit 15 of product 
      V = 0 
      Z = 1 if product is zero; 
              otherwise = D 
      C = 0
SOFTWARE SECTION

MICROCOMPUTER DEVELOPMENT SOFTWARE