Using PCbug11 as a Diagnostic Aid for Expanded Mode M68HC11 Systems

By Steven McAslan,
Motorola Ltd.,
East Kilbride, Scotland

INTRODUCTION

This application note describes some advanced uses of the PCbug11 software package for the M68HC11. The techniques described here allow the user to optimise the debugging environment (perhaps for diagnostic purposes), by moving the communications program into external memory and making full use of the mode programming of the M68HC11. Firstly, the communications routine itself is explained, then the system architecture required is examined and finally the task of customising the talker for the application system is considered. The PCbug11 software is available from Motorola and provides a complex debugging environment for simple hardware platforms.

HOW TALKERS WORK

The PCbug11 environment consists of two pieces of software: the executable on the PC and the communications program which runs on the M68HC11. The communications program is called the talker. The talker is an interrupt-driven and very compact piece of code; either the SCI or XIRQ interrupt can be used.

The purpose of the talker is shown in the flow chart in figure 1. An example of the code used to implement the function is shown in listing 1. This is specifically for the MC68HC11E9. However, the code blocks and label names are normally common to all talkers.

There are three main sections to the code: initialisation; command interpreting; and breakpoint handling. The last two of these are driven by interrupts, while the initialisation is performed only once, whenever the talker is activated.

Briefly, initialisation sets up the internal SCI or external ACIA, enables the appropriate interrupt and ensures that the interrupt vector for this is pointing to the interrupt server, in this case the command interpreter.

The command interpreter has four main functions and two simple communications handlers. The functions are:

- Read Memory (command: $01; label: TREADMEM)
- Write Memory (command: $41; label: TWRTMEM)
- Read Registers (command: $81; label: INH1)
- Write Registers (command: $C1; label: INH2)

The register operations are specific examples of the memory reads and writes, as the register modifications only involve an alteration of the active stack frame in memory.

The functions are selected using the command received. The register commands involve a set number of bytes being transferred from the host to the M68HC11 or vice versa, therefore only a single command byte is required. The memory commands involve communication from the host to instruct the M68HC11 how much memory is to be read/written and the appropriate addresses. For full details, refer to the flow chart and listing software.
Figure 1. TLKTE.ASC Flow Chart

TLKRSTART

INIT STACK
INIT SCI (rx, tx, baud, interrupt)
INIT STOP & I interrupt

IDLE LOOP

SCISRV

Get Data from Rx
Complement and return byte

Inherent (reg) cmd?

YES

NO

Read Registers?

YES

Return SP to Host
Return Stack frame to Host

NO

Get byte count and address

NO

Write Registers?

YES

Get stack pointer from host
Get new stack frame from host

NO

Memory Read?

YES

Memory Write?

NO

SWI reply?

YES

Return breakpoint address to host. Return stack frame to host. Force SWIDLE as return address

Return

Send byte to host Loop Forever
Two communications routines are also used here. These perform reads and writes of the SCI/ACIA (INSCI, OUTSCI). Every command received by the talker is echoed back to the host complemented to confirm communications integrity.

In addition, there is breakpoint handling software. This is more complex, as it involves at least two interrupts to provide full functionality. Before the software can be run, the SWI interrupt vector must be initialised. This is done by the host computer before a go, call or trace command. (See [1], section 4.3.)

The first interrupt occurs when the M68HC11 executes a SWI opcode in its program. This causes a jump to the breakpoint handling software. The SWI interrupt handler then transmits a byte to the host to inform it that an SWI has been found. The M68HC11 enters an idle loop while the PC host determines whether the SWI found is a breakpoint, tracepoint or user SWI. Having decided on the nature of the SWI, the host sends a byte to the MCU to cause the second interrupt. If a user SWI is found, then the code at the user interrupt is simply executed. If a break or trace point is found, then the code suspends at the idle loop until the user decides either to trace again, continue or stop the code execution.

**USING TALKERS IN EXPANDED MODE**

Most users of PCbug11 communicate with a M68HC11 running in bootstrap mode. This involves downloading a talker each time communications begin or using internal EEPROM. However, in embedded systems using an expanded mode M68HC11 it would be more useful to place the talker in external memory with any self-test software. This approach also allows an alternative to the M68HC11 SCI system to be used; a feature which may be useful when the user requires test software running on the SCI.

To use a talker in expanded memory, the basic blocks described in the preceding section must be implemented and the interrupt structure must be able to accommodate the requirements of the talker. The basic blocks are easily moved to an area of expanded memory. However, the interrupt structure does require to be examined quite closely.

The PCbug11/talker environment requires that certain vectors are pointing to certain pieces of code. For trace and breakpoint it is normal for the SWI vector to be altered according to the function in use. In bootstrap mode all of the interrupt vectors point to RAM. From RAM, an appropriate jump to an interrupt service routine can be carried out. This allows the interrupt vectors to be easily customised for the PCbug11 environment. In expanded mode the interrupt vectors point to the top of memory. From here, the user must either redirect them to an writable area of memory or have the block at the top of memory itself writable. Unfortunately, interrupt vectors in RAM would not normally be considered a sound system decision. Such techniques, however, are valuable when developing designs.

The BUFFERO monitor for the M68HC11 redirects the interrupt vectors to internal RAM. In PCbug11 systems, the RESET vector should point to something which will initialise the rest of the vectors in RAM and the talker code. After this the user may load application dependent addresses into the RAM and run his code. The disadvantages here are that there will be a slight processing overhead to reach the interrupt service routine (one extended JMP instruction = 4 cycles) and some 60 bytes of internal or external RAM will be lost to interrupt re-direction. (See [2,3] bootstrap ROM listings. See figure 2 for the memory map arrangement of this system.)

Another approach is to use the special test mode of the M68HC11 MCU. This mode is normally used for factory test purposes, as it allows access to normally protected features of the chip. However, it does have a notable additional feature, which is that the interrupt and reset vectors are transposed from their normal positions in memory at $FF0 to the special mode area $BF0. Note that bootstrap mode also has the same effect. The key difference is that in special test mode the vectors are taken from external memory, rather than the internal bootstrap ROM.

Special test mode could be accessed using a switch or key on the system. The talker interrupt vectors could be placed at the special mode interrupt locations or the interrupt locations could point into RAM; cf. figure 2. In either case, the talker could be placed in some spare area of memory (the talker is normally less than 200 bytes) and only accessed in the special mode.

This approach allows the M68HC11 to be run in expanded mode while retaining the full features of PCbug11. An example of this approach is illustrated in figure 3.
NOTE: Talker uses SCI or XIRQ, SWI initialised by PCbug.

NOTE: Vectors can be redirected to any RAM location decided by user (internal or external).

Talker can be positioned anywhere in memory (internal or external) by user.

Figure 2. Expanded Mode Use Version 1
Figure 3. Expanded Mode Use Version 2
IMPLEMENTING THE EXPANDED MODE TALKER

The following discussion assumes that the user is going to modify an existing talker. If a new talker is to be written, care should be taken that the general principles described in the above sections are adhered to. A general purpose talker for the M68HC11 in expanded mode using the SCI is shown in listing 2.

The first decision to make when implementing a talker in an expanded mode is whether the internal SCI or an external ACIA device is to be used. If the SCI is used, then normally the SCI interrupt or the XIRQ interrupt would be used. It is also possible to use the IRQ interrupt or a timer input capture pin. However, these offer little advantage over the SCI interrupt itself. If an external communications device is used, then the choice is normally the XIRQ interrupt. Again, other interrupt sources can be used, but the XIRQ interrupt should ensure that the communications from the host are responded to.

The use of the XIRQ pin for the external communications device does not prevent the use of the XIRQ for other external resources. If another resource requires to use this pin, then internal arbitration could be used to select which source caused the interrupt. It is essential in this case that there is no possibility of the alternate source causing an endless loop from which the program could never recover.

Once the communications system is chosen and the interrupt to be used is selected, the initialisation section for the talker can be implemented. At this stage any baud, parity, number of bits and interrupt enable bits are set up. It is usually best to perform this function immediately after RESET but it could be performed later if required, for example, if an error is found.

The rest of the talker is not normally changed. However, take care that the M68HC11 registers are not moved using the INIT register and that the INSCI and OUTSCI routines are changed to handle an external device if required.

The last change required is to update the talker .MAP file.

UPDATING THE .MAP FILE

The .MAP file contains essential address information for PCbug11. In bootstrap mode the program knows where certain parameters are by default. However, in expanded mode the talker could be anywhere in memory and so the PCbug11 has to be told where to find it. It is important that the .MAP file corresponds correctly to the talker or malfunction of the software can occur.

Listing 3 shows the .MAP for the general purpose talker in listing 2. The requested addresses may be determined by assembling the talker and noting the location of each of the important labels.

Change the .MAP file using a text editor and place it in the current working directory. The address parameters must begin in the 15th column or higher.

USING THE TALKER AS A DIAGNOSTIC AID

The exact use of the talker in this situation will depend largely on the system which is being examined. However, with the talker installed the user can interactively examine the system. Self-test routines could be run, loaded into RAM from the user PC. EEPROM integrity and preset values could be checked and updated if necessary. If required, the MCU mode could be changed by writing into the HPRIO register. The upper nibble of this register is accessible only in special modes (see [2]).

If the MCU SCI port is available, the device could be placed in special bootstrap mode and PCbug11 run as normal. In this case, the data and address bus integrity of the system could be checked. Here, mode control of the M68HC11 is again the key feature. By changing the HPRIO register (MDA bit), the external data and address buses are turned on while the bootstrap ROM is still present and readable by the CPU. Now the user can perform reads and verifies on the external memory to see if any problem exists with either bus, while still having full control on the MCU via PCbug11.
CONCLUSION

By using the techniques described, the user can include a debugging aid for any expanded mode M68HC11 system. If a single chip system is used, then the additional overhead of PCbug11 RAM requirements is the only drawback.

REFERENCES

[3] MC68HC11 Bootstrap Mode, Motorola Application Note AN1060/D
LISTING 1 – TALKE.ASC ASSEMBLY LISTING

********** TALKE.ASC 6/3/90 **********

* Motorola Copyright 1988,1990
* MCU resident, Interrupt driven Communication routines for 68HC11
* Provides low level memory and stack read/write operations.
* This talker DOES NOT use XIRQ
* Works with Host user interface program PCBUI11.EXE.
* N.B. TALKE.ASC is designed to be downloaded through standard type of
  * bootloader, and communicate with host through SCI.
  * This bootloader relies on 4 char idle line on SCI to terminate.
* Motorola Copyright 1988,1990

* REGISTERS
BAUD equ $2B
SCCR1 equ $2C
SCCR2 equ $2D
SCSR equ $2E
SCDR equ $2F
RDF equ $20
TDRE equ $90

* PROGRAM
IDELE JMP IDELE Wait for SCI interrupt from host.
* A RESET from host changes above jump destination to start of user code.

* CALL
TALKBASE equ $0000
STACK equ $01FF
REGBASE equ $1000
JSCI equ $00C4
JXIRQ equ $00F1
JSMI equ $00F4
JHLOP equ $00F7
JCPF equ $00FA
US500 equ $0080/35
JMPEXT equ $7E
BRKCODE equ $4A
BRACK equ $4A

* CONSTANTS
TALKBASE equ $0000
STACK equ $01FF
REGBASE equ $1000

Init SCI to 9600 baud, no parity
Enable STOP & I bit, disable XIRQ.

On detecting interrupt,
assume receiver caused it.
otherwise program will hang up

Talker code processes data.
Get command byte, & echo as ack
Inverted
To host.
If bit 7 set, process inh. command
else read byte count into B
Save command and byte count.  
Read high address byte into ACCA 
then low address byte into ACCB 
Cmd in A, count in B, addr in X 

If command is memory read, then 

REPEAT 
read required address 
send it to host 
save byte count) 
(restore byte count) 
Increment address 
Decrement byte count 
UNTIL all done 
return to idle loop or user code 

If command is memory write then 

move byte count to ACCA 

REPEAT 

Set up wait loop 
Y operand must be manually set 

Read stored byte and 
end it back to host, 

Increment address 
Decrement byte count 
UNTIL all done 
and return 

If break detected 
then restart talker 

then read data received from host 
and return with data in ACCB 

Only register Y modified. 
Enter with data to send in ACCA. 

MS bit is TDRE flag 

Important - Updates CCR

Move stack pointer to X 
then to ACCD 
send SP to host (high byte first) 
then low byte 
Restore X (=stack pointer) 
then return 9 bytes on stack 
- CCR, B, A, XH, XL, YH, YL, PCH, PCL
If command is write registers then
get SP from host (High byte first)
Move to X reg and copy to stack pointer
Then put next 9 bytes on to stack

Breakpoints generated by SWI
Force host to process breakpoints by sending it the break signal
then wait for response from host.

If host has acknowledged BP then
move SP to SWI stack frame and

Send user code BP return address (high byte first)
(low byte next)
force idle loop on return from BP
but first, return all registers to host

Jump table only during bootstrap
SCI
SPI (Unused vectors point to RTI)
Pulse acc. Input Edge
Pulse acc. Overflow
Timer Overflow

Real Time Intr
IRQ
Real Time Intr
IRQ
SYMBOL TABLE:  Total Entries=39

BAUD     002B   RXSRV    001C
BOOTVECT 0044   RXSRV1   002E
BRKACK   004A   RXSRV2EX 0058
BRKCODE  004A   SCCR1    002C
IDLE     0012   SCCR2    002D
INH1     0075   SCDR     002F
INHIA    0079   SCDRV    0015
INH2     0085   SCSR     002E
INSCI    0059   STACK     01FF
JCOP     009A   SWIDLE    009B
JILLOP   009F   SWISRV    0094
JMPEXT   007E   SWISRV1   009B
JSCI     00C4   TALKBASE  0000
JSWI     00F4   TDRE      0080
JXIRQ    00F1   TLRSTART 0000
NULLSRV  00F8   TREATMEN 0031
OUTSCI   00E0   TWRITEMEN 0043
OUTSC11  006A   WAITFOLL  0048
RDRF     0020   uS500    008E
REGBASE  1000

Total errors: 0
LISTING 2 – TALKSCI.ASC ASSEMBLY LISTING

M68HC11 Absolute Assembler Version 2.70g: talksci.ASC

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

*************** TALKSCI.ASC 14/8/91 ***************

Motorola Copyright 1988, 1991

MCU resident, Interrupt driven Communication routines for 68HC11
* monitor. Provides low level memory and stack read/write operations.

This talker DOES NOT use XIRQ

TALKSCI.ASC is a general purpose talker. It is intended to be
* placed in the MCU memory map at $6000 but this can be changed by
the user to any suitable address. The talker is for general debug
* and can be used in any mode as long as the vectors are correctly
initialised. It is therefore useful for normal modes. The SCI is
* used for communications - use TALKACIA when an external ACT is
* to be used. TALKSCI assumes that the interrupt vectors are
* pointing to RAM in the same way as the bootstrap ROM.

IMPORTANT : If you change the running address of this program
then you MUST also change the TALKSCI.MAP file so that the two
match.

When PCBUG11 is executed with option TALKSCI, a 102D break is
output to the 68HC11's SCI, prior to establishing communication.

CONSTANTS
TALKBASE equ $6000
STACK equ $003F
BOOTVECT equ $00C4
REGBASE equ $1000

JSCI equ $00C4
JXIRQ equ $00F1
JSMI equ $00F4
JILOP equ $00F7
JCOF equ $00FA

JMPEXT equ $7E
BKRCODE equ $4A
BKACK equ $4A

BAUD equ $2B
SCCR1 equ $2C
SCCR2 equ $2D
SCSR equ $2E
SCDR equ $2F

RDF equ $20
SEP equ $80
OR equ $08
PE equ $02

* REGISTERS
organic TALKBASE

* PROGRAM
TALKSTART equ Dynamically set up Boot jump table.

LDAA #JMPEXT
LDY #NULLSEV
LDA #BOOTVECT
STA .X
INX
STY .X
INX

MOTOROLA
Initialise SCI to 9600, no parity
and enable SCI tx & rx:
Enable STOP, 1 interrupts, disable X

User may add jump to his own code here or may move the above
initialisation to the start of his own program.

Now hang around for SCI interrupt
On detecting interrupt,
assume receiver caused it.
otherwise program will hang up
Talker code processes rec'd data.
Inverted.

Repeat
read required address
send it to host

If bit 7 set, then process inh cmd
else read byte count from host into B

Read high address byte
into ACCA
then low address byte into ACCB
Cmb in A, count in B, addr in X

If command is memory read, then
Read stored byte and
echo it back to host,

If command is memory write then
move byte count to ACCA

Read next byte from host into ACCB,
and store at required address.
Set up wait loop
Y may take on suitable value

Read stored byte and
echo it back to host,

INSCI EQU *  
LDAB SCSR+REGBASE  Wait for RDRF=1
BITB # (FE+OR)  
If break detected then
BNE TLKRSTART  restart talker.
ANDB # RDRF
BREQ INSCI
LDAB SCDR+REGBASE  then read data received from host
RTS
and return with data in ACCB

OUTSCI is the subroutine which transmits a byte from the SCI

OUTSCI EQU *  
Only register Y modified.
LDAB SCSR+REGBASE
XDDY
Enter with data to send in ACCA.

OUTSCI1  
LDDA SCSR+REGBASE
PPL OUTSCI1
MS bit is TDRE flag

INSCI EQU *  
SCSR+R~BASE  Wait for RDRF=1

# (FE+OR)  
if break detected then
TLKRSTART  restart talker.
#RDRF
INSCI
SCDR+REGBASE then read data received from host
and return with data in ACCB

Command was to read MCU registers

INMA  
TSX  
Move stack pointer to X
XDDX
then to ACCD

INMB  
XDDX
BSR OUTSCI
Send SP to host (high byte first)

INMC  
TRI
BSR OUTSCI
then low byte

INMD  
LDDA #9
Then return 9 bytes on stack

INME  
BRA TREDMEM
i.e CCR, R, A, XH, XL, YH, YL, PCH, PCL

Command was to write MCU registers

INMF  
CMPA #37E
If command is read registers then
BNE INM1

INMG  
XDDX
BSR OUTSCI
get SP from host (high byte first)

INMH  
XDDX
BSR INSCI
get low byte next

INMI  
XDDX
Move to X reg

INMJ  
XDDX
and copy to stack pointer

INMK  
LDDA #9
Put next 9 bytes from host onto stack

BRA TWRITMEM

An SWI interrupt was generated

SWISR EQU *  
Breakpoints generated by SWI
LDAB #BBR CODE
Force host to process breakpoints
BRA OUTSCI  by sending it the break signal

SMIDLE is the infinite loop which allows the 'STOPPED' mode of PChug

SWIDLE CLI
BRA SWIDLE
then wait for response

If host acknowledged then
202 A 60BD 26B9  BNE RXSRVEX
203 A 60BF 30  TXS
204 A 60C0 C609  LOAB #9
205 A 60C2 3A  AXS
206 A 60C3 35  TXS
207 A 60C4 B607  LDD 7,X
208 A 60C6 8600  BSR OUTSCI
209 A 60C817  TBA
210 A 60C9 88BD  BSR OUTSCI
211 A 60CB CC60B8  LDD #SWI_IDLE
212 A 60CE ED07  STD 7,X
213 A 60D0 20C7  BRA INH1A
214 A
215 A  END

SYMBOL TABLE:  Total Entries=42

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUD</td>
<td>002B</td>
<td>REGBASE</td>
</tr>
<tr>
<td>BCOVFCT</td>
<td>00C4</td>
<td>RXSRV</td>
</tr>
<tr>
<td>BRACK</td>
<td>004A</td>
<td>RXSRV1</td>
</tr>
<tr>
<td>BRECODE</td>
<td>004A</td>
<td>RXSRVEX</td>
</tr>
<tr>
<td>FE</td>
<td>0002</td>
<td>SCCR1</td>
</tr>
<tr>
<td>IDLE</td>
<td>6032</td>
<td>SCCR2</td>
</tr>
<tr>
<td>INH1</td>
<td>6095</td>
<td>SCSR</td>
</tr>
<tr>
<td>INH1A</td>
<td>6099</td>
<td>SCISRV</td>
</tr>
<tr>
<td>INH2</td>
<td>60A5</td>
<td>SCSR</td>
</tr>
<tr>
<td>INSCI</td>
<td>6079</td>
<td>SETVECT</td>
</tr>
<tr>
<td>JCOP</td>
<td>00FA</td>
<td>STACK</td>
</tr>
<tr>
<td>JILLOP</td>
<td>00F7</td>
<td>SWIIDLE</td>
</tr>
<tr>
<td>JMPEXT</td>
<td>007E</td>
<td>SWISRV</td>
</tr>
<tr>
<td>JSCI</td>
<td>00C4</td>
<td>SWISRV1</td>
</tr>
<tr>
<td>JSWI</td>
<td>00F4</td>
<td>TALKBASE</td>
</tr>
<tr>
<td>JXIRQ</td>
<td>00F1</td>
<td>TDRS</td>
</tr>
<tr>
<td>NULLSRV</td>
<td>6078</td>
<td>TLRSTAR</td>
</tr>
<tr>
<td>OR</td>
<td>0008</td>
<td>TREADMEM</td>
</tr>
<tr>
<td>OUTSCI</td>
<td>6088</td>
<td>TWTIMEM</td>
</tr>
<tr>
<td>OUTSCI1</td>
<td>608A</td>
<td>USERSTAR</td>
</tr>
<tr>
<td>RDRF</td>
<td>0020</td>
<td>WAITPOL</td>
</tr>
</tbody>
</table>
LISTING 3 – TALKSCI.MAP

Name of constant must not exceed 14 characters.
Value of constant must start in column 15 or higher.

talker_start $6000  Talker code start address.
talker_idle   $6032  Talker code idle loop address.
user_start   $6020  User’s reset entry into talker code.
xlim_qjmp     $00F2  Address in talker code of user’s XIRQ server address.
re locate bu f $00A0  Address to where user’s code is relocated on break point.
xirq_xrv      $6035  Talker’s XIRQ service address.
swi_xrv      $60B4  Talker’s SWI service address for break points.
swi_idle     $60B8  Talker’s SWI idle loop.
null_xrv     $6078  Talker RTI.
xirq_jmp     $00F2  XIRQ vector.
swi_jmp      $00F5  SWI vector.
cma_jmp      $00FB  COP clock monitor vector.