Command Prompt Shell for M68HC11E1

Lab #2

Junsang Cho
Feb. 4, 2003
**Command Prompt Shell for M68HC11E1**

1. **Objective**
   - Learn how to use a simple simulator for the 68hc11
   - Gain experience with interrupt handlers and serial interface drivers.
   - Complete a 68hc11 command prompt shell that will be useful for future labs

2. **Tasks**
   a. Simulate the count_a.asm program from lab1 using the Wookie simulator.
   b. Assemble, load, and run the lab2.asm program. Use the oscilloscope to observe characters as they are echoed.
   c. Extend the SCI receive interrupt handler in the program lab2.asm to buffer characters. Use carriage-returns and a max buffer size of 165 to terminate commands. Interrupt “U”, “D” and “Q” command to make the LED counter count up, count down, and turn off, respectively.

3. **Tools**
   - RS232/TTL converter cable.
   - D25-D9 adapter
   - Bench cable set
   - AS11.exe assembler
   - DL11.exe loader
   - Load_e1.bat batch file
   - Test program to count on port A
   - Two customized programs

4. **Resources**
   - Kit for lab1
   - The Wookie 68HC11 simulator
   - Hyperterminal
   - RS232/TTL converter cable
   - AS11 assembler
   - DL11 loader
   - Load_e1.bat batch file
   - Test program to count on port A

By Junsang Cho
5. Procedures

a. Set up the Wookie simulator to run the count_a.asm program.
b. Run and test the lab2.asm program on my 68HC11 circuit.
c. Extend both channels of the oscilloscope; probe the RxD and TxD signals on the 68hc11. Observe the delay between the characters received by the 68hc11 and the echoed characters. Type the number 0 through 9, and notice how the pattern on the oscilloscope changes.

6. Write Up

a. Encountered problems
   - We misunderstood about “bcrl” and “bset”. So code was screw up.
   - We recognized that “#count_on + count_up” was not worked later.
   - We couldn’t distinguish between “#label” and “label”. So code was screw up.

b. How might the lab2.asm program fail if the instruction “lds #STACK” were omitted?
   → No effect anything in code.

c. In the initsci initialization code in lab2.asm, what would happen if the instruction “ldaa #$2c” was replaced with “ldaa #$ac”?
   →
   
   | SCCR2 | TIE | TCIE | RIE | ILIE | TE | RE | RWU | SBK | $102D |
   |-----------------------------------------------|
   | Actually, “ldaa #$2c” is enable RIE, TE and RE bit. But if I put “ldaa #$ac” is also enable TIE bit. It means that the interrupt will occur on both input stream and output stream. Hence it will screw up my code.

   d. Explain why the instruction, “brclr SCSR,Y #SCSR_TDRE txchr_” at the beginning of the txchr interrupt handler is necessary. Will this interrupt handler even be called if the TDR is still full? Explain.
   → TDRE(Transmit Data Register Empty Flag) set if transmit data can be written to SCDR; if TDRE is 0, transmit data register contains previous data that have not yet been

By Junsang Cho
moved to the transmit shift register. Writing into the SCDR when TDRE is set will result in a loss of data.

Also the interrupt handler is always called because it must check TDRE bit, however, if you put a control variable for this, you can make it to work only necessary.
Appendix. Code detail

******************************************************************************
* A serial command prompt shell for the 68HC11
* History
* wep, 020120, Completed prompt with commands U, D, Q.
* Removed some of the code for rxchar.
******************************************************************************
******************************************************************************
* Conventions
* * IY points to the register base ($1000) to support efficient register access. Using IY is more space and time efficient for
* bit manipulation and testing of memory, but loads and stores are more efficient with extended addressing. We might
* want to consider using IX for register addressing, since this saves a pre-byte with every instruction. It depends on what
* else we need IX for.
* * Currently, functions do not save registers to the stack.
******************************************************************************
******************************************************************************
* Equates
******************************************************************************

* Registers

REGBS    EQU   $1000 start of registers
BAUD     EQU   $2B sci baud reg
SCCR1    EQU   $2C sci control1 reg
SCCR2    EQU   $2D sci control2 reg
SCSR     EQU   $2E sci status reg
SCDR     EQU   $2F sci data reg
PORTA    EQU   $00
PACTL    EQU   $26 Port A control reg.
OC1M     EQU   $0C Port A alternate function reg.s
OC1D     EQU   $0D ""
TCTL1    EQU   $20 ""

* Register fields and bit masks

SCCR2_TIE EQU   $80 transmit interrupt enable
SCSR_TDRE EQU   $80 transmit data register empty
SCSR_RDRF EQU   $20 receive data register full

LED_ON   EQU   $01 LED count enable
LED_UP   EQU   $02 LED count up

* Other equates

JMP_OP   EQU   $7E Operand for JMP instruction
CARRIAGE_RET EQU   $0D
LINE_FEED EQU   $0A
ASCII_D  EQU   $44
ASCII_U  EQU   $55
ASCII_Q  EQU   $51

******************************************************************************
* Stack
* * The stack grows downward from the last byte before the pseudo-vector jump table.
******************************************************************************

STACK    EQU   $00C3

******************************************************************************
* Uninitialized variables
* * These will be placed in the first 256 bytes of internal RAM to support direct addressing. They must be explicitly initialized
* by the code.
******************************************************************************

ORG     $0000

By Junsang Cho
* Transmit buffer
  TXPTR RMB 2  address of next character to transmit
  RXBUF RMB 16  up to 16 bytes for commands
  RXBUF_ RMB 1  plus one byte for null-termination
  RXPTR RMB 2  pointer to next write position

* LED control bits
  * bit 0: 0 = LEDs off, 1 = LEDs count
  * bit 1: 0 = count down, 1 = count up
  LEDCTRL RMB 1  LED control bits

* Vector jump table ($00C4)
* These jump commands must be installed at run time, since this memory is used by the talker program for download.

<table>
<thead>
<tr>
<th>ORG</th>
<th>$00C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSCI</td>
<td>RMB 3</td>
</tr>
</tbody>
</table>

* Entry point for program. Initialize stack and pseudo-vectors.

<table>
<thead>
<tr>
<th>ORG</th>
<th>$B600</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td></td>
</tr>
</tbody>
</table>
  lda #STACK  set the stack pointer
  ldy #REGBS  use IY for accessing registers
  ldaa #JMP_OP set the bootstrap mode SCI pseudo-vector
  staa JSCI  JSR instruction
  ldx #int_sci address of interrupt handler
  stx JSCI+1

* Initialize port A for LED test.

| lda  | #$80  | enable PA7 for output
| staa | PACTL,Y | disable PA alternate functions
| clra | OC1M,Y |
| staa | OC1D,Y |
| staa | TCTL1,Y |

* Initialize SCI.
  * 9600 baud at 8 MHz Extal.
  * Enable transmitter, receiver, and receive interrupt.

| ldaa | #$30  | baud register (m=0; wake=0)
| staa | BAUD,Y |
| clra | SCCR1,Y |
  ldaa #$2c  enable (~TIE | RIE | TE | RE)
  staa SCCR2,Y
  ldx #NULLCHR make TXPTR point to a permanent NULL
  stx TXPTR ...until txstr is called
  ldx #RXBUF initialize RXPTR -> RXBUF
  stx RXPTR
  cli  enable maskable interrupts

* Other initialization. Enable LEDs to count up.
  * We don't transmit an initial banner or prompt, since our circuit comes up
  * with the serial port disconnected.

| bclr | LEDCTRL #$f |

By Junsang Cho
bset LEDCTRL $#3  set LED control for counting up

* Main loop
* A simple, example foreground task. Just count on four LEDs.
* Serial processing is all interrupt driven.

******************************************************************************
* Main loop
******************************************************************************
test jsr ledtst  count on the upper nibble of Port A
ldx $#ffff  dela y value
jsr delay
bra test

******************************************************************************
* Subroutines
******************************************************************************

******************************************************************************
* LED test. On each pass, update the 4-bit value displayed in the upper
* nibble of port A. Behavior depends on the LEDCTRL byte.
******************************************************************************
ledtst ldaa PORTA,Y  load current LED pattern
bset LEDCTRL #LED_ON led_on  test whether count enabled
cira ledtst1
bra led_on  if not enabled, clear PORTA
brclr LEDCTRL #LED_UP led_dn  test for up or down count
adda #$10  increment upper nibble of A
bra ledtst1
suba #$10  decrement upper nibble of A
ledtst1 staa PORTA,Y
rts

******************************************************************************
* Delay. Use the value in IX as a delay count.
******************************************************************************
delay dex
bne delay
rts

******************************************************************************
* Transmit string
******************************************************************************

bset TXPTR save address of transmit string
bset SCCR2,Y #SCCR2_TIE enable int's for empty tx data register

******************************************************************************
* Transmit character
******************************************************************************

By Junsang Cho
ECE3720 Embedded Microcomputer System

Lab Stick Report

---

**txchr**

*brclr SCSR,Y #SCSR_TDRE txchr_ return if TDR still full*

*ldx TXPTR load U16 address of next character to IX*
*ldaa 0,X load A with next character*
*beq stoptx terminate transmit if NULL character*
*inx otherwise, increment TXPTR*
*stx TXPTR ...and update*
*staa SCDR,Y load character (still in A) for transmission*
*bra txchr_ *

**stoptx**

*bcclr SCCR2,Y #SCCR2_TIE don't let empty TDRE cause interrupt*

*ldx #NULLCHR make TXPTR point to a permanent NULL*
*stx TXPTR ...until txstr is called again*

**txchr_**

*rts*

'* Receive character*

'* If receive data register empty, return.*
'* Otherwise, read and echo chr.*
'*
'* PENDING! Add code to implement the following pseudo-code.*
'*
'* If received character == <CR> or RXBUF full,*
'* call do_cmd to process command and display prompt*
'* else*
'* put new character in RXBUF,*
'* update RXPTR++.*
'*
'* Modified by Junsang Cho on Jan.28, 2004*

*-------------------------------*

**rxchr**

*brclr SCSR,Y #SCSR_RDRF rxchr_ return if RDRF empty*

*ldaa SCDR,Y read newly received character*
*staa SCDR,Y echo char*

***** PENDING! Add code described in rxchr function header. *****

* cmpa     #CARRIAGE_RET I f received character == <CR>, goto clr*
* beq      CLEAR*
* ldx      RXPTR update RXPTR++*
* staa     RXBUF put new character in RXBUF*
* cpx      #RXBUF_*
* beq      CLEAR*
* inx      stx RXPTR bra rxchr_*

**CLEAR**

*ldx #RXBUF_ Initialize RXPTR -> RXBUF*
*stx RXPTR*
*jsr do_cmd call do_cmd to process command*

**rxchr_**

*rts*

*-------------------------------*

* do_cmd*

* Invoke appropriate command here based on the contents of RXBUF*
* Currently, we only examine the first character of RXBUF.*
* Eventually, commands will contain arguments.*

*-------------------------------*

*do_cmd ldax RXBUF load A with the first command character*
* U: (Up) Set the LEDCTRL:LED_UP bit for counting up; clear for counting down.

```assembly
do_cmd1 cmpa #ASCII_U check for "U" command
  bne do_cmd2
  bclr LEDCTRL #$f
  bset LEDCTRL #$3 Set LEDCTRL for counting up
  bra prompt valid command, display prompt
```

* D: (Down) Clear the LEDCTRL:LED_UP byte for counting down.

```assembly
do_cmd2 cmpa #ASCII_D check for "D" command
  bne do_cmd3
  bclr LEDCTRL #$f
  bset LEDCTRL #$1 Set LEDCTRL for counting down
  bra prompt valid command, display prompt
```

* Q: (Quit) Turn off LEDs and don't count

```assembly
do_cmd3 cmpa #ASCII_Q check for "Q" command
  bne badcmd
  bclr LEDCTRL #$f
  bset LEDCTRL #$00 Set LEDCTRL for counting stop
  bra prompt
```

* Add future commands here. When a command is not recognized, execution falls
* through to the "badcmd" label, and an appropriate message is displayed.

```assembly
badcmd ldx #BADPROMPT Display possible cmds + prompt for invalid cmd
  jsr txstr
  bra do_cmd_ ...and return
```

* Display next prompt here.

```assembly
prompt ldx #GOODPROMPT Display prompt for valid cmd
  jsr txstr
  do_cmd_ rts end of do_cmd
```

* Interrupt handlers

* SCI

```assembly
int_sci jsr txchr call the transmit character routine
  jsr rxchr call the receive character routine
  rti
```

* Constants

* Note that the bad prompt includes the good prompt, but adds a prefix.

BADPROMPT
```
FCB CARRIAGE_RET Response to invalid cmd = "\n?r\nHC11> "
FCB LINE_FEED
FCC "Commands: U, D, Q"
```

GOODPROMPT
```
FCB CARRIAGE_RET Response to valid cmd = "\nHC11> 
FCB LINE_FEED
FCC "HC11> "
```

NULLCHR FCB 0 null-termination and safe place for the TX ptr