Introduction to the Motorola 68HC908JB8

Jean-Michel FRIEDT, December 2001-October 7, 2004

1 Introduction

We decided to look at the 68HC908JB8 microcontroller from Motorola following interest in the development of USB enabled intruments. The 68HC908JB8 does not require any external programmer since it is based on flash memory, provides a convenient communication tool in ROM called the Monitor mode, includes one USB peripheral but no UART. It is available both is easy to handle DIP package and low volume SOIC package.

All development were done under Linux (kernel 2.2.19, although no kernel specific functions were used) using `gcc` 2.95.2 for generating the binaries running on the PC and `as6808` v.03.10 provided by `asxxxx` v.3.10 (November 2001) for the programs running on the microcontroller.

The following script was used for generating the ASCII file containing the hexadecimal codes of the program to be transferred from the PC to the microcontroller:

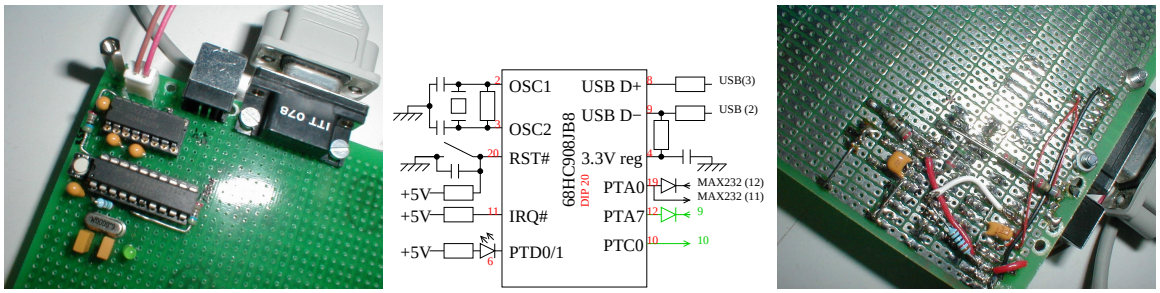
```
#!/usr/bin/tcsh
if ( $# == 0 ) then                     # requires at least 1 argument
  echo $0 "filename[.asm]"
else
  set nom=$1                            # remove extension from name (if given)
  if ( `echo $1 | grep \.` != "" ) then
    set nom=`echo $1 | cut -d\. -f1`
  endif
  echo $nom "->" $nom".out"             # input foo.asm outputs as foo.out
  ./as6808 -o $nom.asm
  grep ^T $nom.rel | cut -c9-80 > $nom.out # keep only data
  \rm $nom.rel
endif
```

`asm`: script for compiling a text file containing the assembly program to an ASCII file containing the hexadecimal opcodes.

Once the `.out` ASCII file containing the hexadecimal opcodes is generated, the board is switched on and the program is uploaded using `./hc08 file.out` where `hc08` is a program described later developed for the purpose of learning how this microcontroller works. But first we must look at the hardware before being able to test our first program.

2 Electronic circuit

The electronic circuit around the 68HC908JB8 is quite simple and highly inspired from the development circuit provided by Motorola. It mainly consists of the oscillator circuit, pull up resistors to the interrupt (IRQ#) and PTA0 lines, and the reset switch. Since the PTA0 line, which is used for communication in the monitor mode with the PC through the MAX232, includes its own internal pull up resistor, adaptation between the TTL (5 V) logic of the MAX232 and the 3.3 V logic of the 68HC908JB8 is quite simple: a diode turned towards the highest voltage is enough. Indeed, if the MAX232 outputs a low level (pin grounded), the current can flow from the microcontroller to the MAX232 and PTA0 senses a ground level. If on the other hand the MAX232 pin is high (+5 V), the diode is blocking the current flow from the microcontroller to the MAX232 and the PTA0 pin senses a high (+3.3 V) level thanks to its pull up resistor.



Simple circuit for communicating with the 68HC908JB8 microcontroller. The green lines are optional connections between the microcontroller and the MAX232 in order to use a second software emulated UART. The red numbers are the pin numbers for a 20 pin DIP package.

3 The Monitor mode

First, we must find out how to program the 68HC908 and get familiar with its monitor feature. To this date (December 2001-January 2002) one program exists for programming two of the 68HC908 family: `spgmr08`, version 0.9. However, this software aims at integrating a lot of features in one bulky executable, including a GUI, which is not what I was looking for. And anyway, understanding every steps of the programming part of the microcontroller is interesting. So after building a basic board including a CPU, a MAX232 RS232 level converter and a few passive components as described earlier, I started putting together a few routines for getting familiar with the monitor mode.

Two tricks appeared:

- when sending the 8 security bytes at the beginning of the transmission, a delay between the received echo and the new

transmission is required
- the echo does not include one but two characters: the direct connexion through the protection diode of the RS232 transmission line with its reception line, followed by the echoed character by the microcontroller.

The monitor mode is otherwise implemented as described in section 10 of the Technical Data book. It allows reading and writing individual bytes or sequentially to any place in the microcontroller's memory, including to the I/O ports which makes testing simple circuits very easy. As a first example, let us make an LED connected to port D pin 0/1 (on the 20 pin DIP package) blink under computer control. We here write to the port D register from the PC: no program is running on the microcontroller itself (apart from the Monitor routine provided in ROM).

```
/* All examples have been derived from miniterm.c                        */
/* Don't forget to give the appropriate serial ports the right permissions */
/* (e. g.: chmod a+rw /dev/ttyS0)                                        */

#include "rs232.h"

extern struct termios oldtio,newtio;

int init_rs232(int BAUDRATE)
{int fd;
 fd=open(HC11DEVICE, O_RDWR | O_NOCTTY );
 if (fd <0) {perror(HC11DEVICE); exit(-1); }
 tcgetattr(fd,&oldtio); /* save current serial port settings */
 bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */
// newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
 newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;  /* _no_ CRTSCTS */
 newtio.c_iflag = IGNPAR; //  | ICRNL |IXON;
 newtio.c_oflag = IGNPAR; // ONOCR|ONLRET|OLCUC;
// newtio.c_lflag = ICANON;
// newtio.c_cc[VINTR]    = 0;     /* Ctrl-c */
// newtio.c_cc[VQUIT]    = 0;     /* Ctrl-\ */
// newtio.c_cc[VERASE]   = 0;     /* del */
// newtio.c_cc[VKILL]    = 0;     /* @ */
// newtio.c_cc[VEOF]     = 4;     /* Ctrl-d */
 newtio.c_cc[VTIME]    = 0;     /* inter-character timer unused */
 newtio.c_cc[VMIN]     = 1;     /* blocking read until 1 character arrives */
// newtio.c_cc[VSWTC]    = 0;     /* '\0' */
// newtio.c_cc[VSTART]   = 0;     /* Ctrl-q */
// newtio.c_cc[VSTOP]    = 0;     /* Ctrl-s */
// newtio.c_cc[VSUSP]    = 0;     /* Ctrl-z */
// newtio.c_cc[VEOL]     = 0;     /* '\0' */
// newtio.c_cc[VREPRINT] = 0;     /* Ctrl-r */
// newtio.c_cc[VDISCARD] = 0;     /* Ctrl-u */
// newtio.c_cc[VWERASE]  = 0;     /* Ctrl-w */
// newtio.c_cc[VLNEXT]   = 0;     /* Ctrl-v */
// newtio.c_cc[VEOL2]    = 0;     /* '\0' */
 tcflush(fd, TCIFLUSH);tcsetattr(fd,TCSANOW,&newtio);
// printf("RS232 Initialization done\n");
 return(fd);
}

void sendcmd(int fd,char *buf)
{unsigned int i,j;
 if((write(fd,buf,strlen(buf)))<strlen(buf))
    {printf("\n No connection...\n");exit(-1);}
 for (j=0;j<5;j++) for (i=0;i<3993768;i++) {}
 /* usleep(attente); */
}

void free_rs232(int fd)
{tcsetattr(fd,TCSANOW,&oldtio);close(fd);} /* restore the old port settings */
```

`rs232.c`: basic RS232 initialization routine needed for all programs running under Linux requiring access to the serial port.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>                 /* declaration of bzero() */
#include <fcntl.h>
#include <termios.h>

int init_rs232();
void free_rs232();
void sendcmd(int,char*);
struct termios oldtio,newtio;

// #define BAUDRATE B9600
// #define BAUDRATE B19200
#define HC11DEVICE "/dev/ttyS0"
```

`rs232.h`: header for the basic RS232 initialization routines.

```
// test hardware: make a diode blink from monitor (MON) mode

#include "hc08.h"

void hc08_test(int fd)
{char lo,hi;
 lo=0x56; hi=0; read_hc08(fd,hi,lo);
 lo=0x57; hi=0; read_hc08(fd,hi,lo);
 lo=0x58; hi=0; read_hc08(fd,hi,lo);
 lo=0x56; hi=0; read_hc08(fd,hi,lo);iread_hc08(fd);
 lo=0x57; hi=0; writ_hc08(fd,hi,lo,0xaa);
 lo=0x58; hi=0; writ_hc08(fd,hi,lo,0x55);
 lo=0x56; hi=0; read_hc08(fd,hi,lo);iread_hc08(fd);
 iwrit_hc08(fd,0x32);
 lo=0x59; hi=0; read_hc08(fd,hi,lo);
 readsp_hc08(fd);
 lo=0x07; hi=0; writ_hc08(fd,hi,lo,0xFF);
 while(1) {
   lo=0x03; hi=0; writ_hc08(fd,hi,lo,0xFF);
   sleep(1);
   lo=0x03; hi=0; writ_hc08(fd,hi,lo,0x00);
   sleep(1);
  }
}

void hc08_prg(int fd)
{char lo,hi;
 lo=0x00; hi=0x01; writ_hc08(fd,hi,lo,0x00);
 iwrit_hc08(fd,0x32);
 readsp_hc08(fd);
}

int main(int argc,char **argv)
{int fd;
    fd=init_rs232(B9600);
    init_hc08mon(fd);free_rs232();
    fd=init_rs232(B9600);// forget the 10 stop bits
    hc08_test(fd);
    free_rs232();
}
```

Test programming for controlling the blinking of an LED connected to port D pin 0/1 (20 pin DIP package) controlled from a Linux running PC.

Once we have checked the basic circuitry around the microcontroller is operating properly, we can go on to the next step of performing the same task (blinking an LED) from a program stored on board the microcontroller. The slight additional difficulty is to figure out how to setup the memory in the microcontroller before sending a `RUN` command to the monitor. I found the answer in a comment to `spgmr08`, in `mongp32.c` (comment to the routine `mon_runpc()`): after asking the monitor the current position of the stack pointer (SP), we simply store in SP+4 the high byte of our program's starting address and in SP+5 the lower byte of our program's starting address. Since I have decided to store my program in the beginning of RAM space which starts at 0x0040, (SP+4)=0x00 and (SP+5)=0x40 in my case. After reset, the monitor mode defines the stack pointer to be located at 0x00FF, and uses a few bytes so that just before executing the `RUN` command, SP always appears to be equal to 0x00FA (=250d).

3.1 Storing and executing a program from RAM

```
#include "hc08.h"

void hc08_prg(FILE *f,int fd)
{char lo,hi;int i=0,j,status;
 fscanf(f,"%x",&j); lo=0x40; hi=0; writ_hc08(fd,hi,lo,(char)j);
 do {DEBUG(" -- %x \n",ctoi(j));status=fscanf(f,"%x",&j);
     iwrit_hc08(fd,(char)j);i++;}                          /* read while !EOF */
     while (status!=EOF);
 DEBUG (" -- %x \n",ctoi(j));printf(" %d bytes sent\n",i);
       /* setup stack for RUN : read SP, store PC in SP+4 (lo) and SP+5 (hi) */
 j=readsp_hc08(fd); DEBUG(" -- SP=%d\n",j);
 lo=((j+5)&0xff);hi=(((j+5)>>8)&0xff);writ_hc08(fd,hi,lo,0x40); // PC LO
 lo=((j+4)&0xff);hi=(((j+4)>>8)&0xff);writ_hc08(fd,hi,lo,0x00); // PC HI
 run_hc08(fd);
 DEBUG("\n RUN executed \n");
}

int main(int argc,char **argv)
{int fd;FILE *f;
 if (argc<2) {printf("%s filename\n",argv[0]);} else {
    fd=init_rs232();
    init_hc08mon(fd);free_rs232();fd=init_rs232();// forget the 10 stop bits
    f=fopen(argv[1],"r");
    hc08_prg(f,fd);
    fclose(f);
    free_rs232();
   } return(0);
}
```

Store a program in RAM (address 0x0040) and execute it.

```
start: ldhx #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov  #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f
clra ; clear accumulator
mov  #0x0f,0x0007 ; DDRD: port D as output

loop: eor  #0x0f ; toggle diode
sta  0x0003 ; store accumulator on PortD
bsr  delay
bra  loop

delay: psha
pshx
clrx ; 256*0,9375ms=240ms
loopx: clra ; 9*256=2304 (0,9375ms @ 2,4576MHz)
loopa: nsa ; [3]
nsa ; [3]
dbnza loopa ; [3]
dbnzx loopx ;
pulx
pula
rts
```

`blink.asm`: sample program for blinking a LED connected to PTD0/1.

The limitation of this programming method is quickly obvious: since the default location of the stack pointer (SP) is 0x00FF, the RAM is cut in two halves (0x0040 to 0x00FA approximately, and 0x00FF to 0x013F) which only allows uploading to the microcontroller programs 186 and 64 bytes long respectively (depending whether we store the program below or above the stack pointer). Since our aim is USB development and the sample (short) program from Motorola already needs 1.8 KB, being able to store the code to flash memory and execute it from there seems mandatory. Our next steps will thus be to develop a software UART (since the 68HC908JB8 does not include an hardware UART) so we can receive new data from our own programs, and then to learn how to store data (the new program we want to save) in flash memory.. This way, we will be able to test programs up to 8 KB long, which should be enough to get us started with USB development.

4 Asynchronous communication (software emulation)

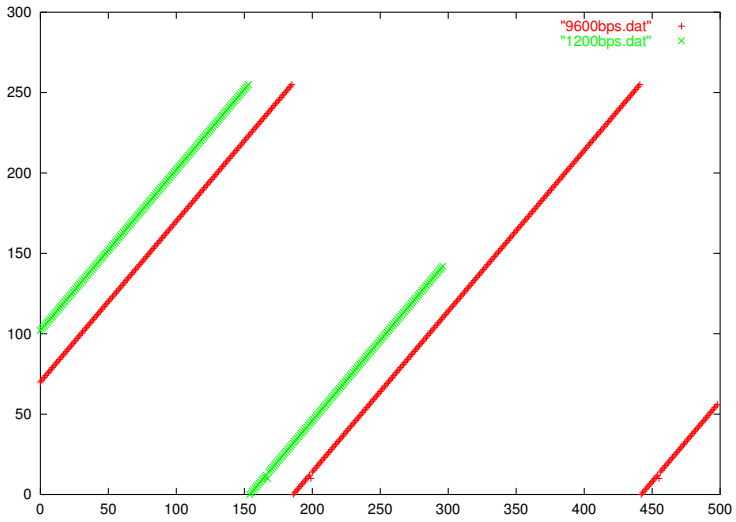
First, we wish to transmit characters from the microcontroller to the PC.

```
start: ldhx  #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov   #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f

mov   #0x00,0x0003 ; PTD0/1 lo => LED lit
mov   #0x03,0x0007 ; DDRD: PTD0/1 as output

mov   #0x01,0x0000 ; PTA: PTA0 hi
mov   #0x01,0x0004 ; DDRA: PTA0 as output
ldx #0h00
loop: incx ; increment counter
txa
bsr  send ; send value of counter to serial port
bra  loop

send: pshx
ldx #0x08 ;   snd through PTA0 the content of Acc (@9600)
mov   #0x00,0x0000 ;   PTA: PTA0 lo : START bit
looprs: bsr delay ; X
bsr delay ; X
rora ; 1 rotate right Acc through carry
bcc bit0 ; 3 branch if carry is clear (is A&1=0)
mov #0x01,0x0000 ; 4 PTA0=hi
bra bit1      ; 3
bit0: mov #0x00,0x0000 ; 4 PTA0=lo
bit1: dbnzx looprs ; 3   --> sum=11 or 14
fin: bsr delay
bsr delay
mov   #0x01,0x0000 ; PTA: PTA0 hi : STOP bit
bsr delay
bsr delay
pulx
rts

delay: pshx ; 2+4 for bsr   (12+2*((X*9)+15))*.3333=833
ldx #0h0f ; 3                                      104
loopx: nsa ; 3    => Xinit=0x88 for 1200
nsa ; 3            =0x0f for 9600
dbnzx loopx ; 3
pulx ; 2
rts ; 4
```

`rs_snd.asm`: sample program for sending the values of a free running counter to the serial port. The baud rate is defined by the delay value in the function `delay` (0x88 for 1200 baud, 0x0f for 9600 baud communication).

```
#include "rs232.h"

void read_osc(int fd)
{unsigned char buf;
 while (1) {read(fd,&buf,1);
   printf ("%u($%x) \n",buf&0x000000FF,buf&0x000000FF);
   fflush(stdout);}
}
void main(int argc,char **argv)
{int fd;
 fd=init_rs232();
 read_osc(fd);
 free_rs232();
}
```

`hc08rec.c`: sample program for reading the values on the RS232 port and displaying their decimal and hexadecimal values.



Result of transmitting the values of a free running counter (increasing) at 1200 (green) and 9600 (red) bauds. All data were correctly transmitted from the microcontroller running the software UART to the PC.

Now that we understand how to transmit arbitrary values from the microcontroller to the PC, we also want the microcontroller to be able to read values from the serial port and process them (for example add 3 and send the result back).

```
start: ldhx  #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov   #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f

mov   #0x00,0x0003 ; PTD0/1 lo => LED lit
mov   #0x03,0x0007 ; DDRD: PTD0/1 as output

; ldx #0h00

loop: mov   #0x00,0x0004 ; DDRA: PTA0 as output
bsr rcv
; incx ; increment counter
; txa
inca
inca
inca
mov   #0x01,0x0000 ; PTA: PTA0 hi (in order to avoid glitches)
mov   #0x01,0x0004 ; DDRA: PTA0 as output
bsr  send ; send value of counter to serial port
bra  loop

send: pshx
ldx #0x08 ;   snd through PTA0 the content of Acc (@9600)
mov   #0x00,0x0000 ;   PTA: PTA0 lo : START bit
loopsn: bsr delay ; X
bsr delay ; X
rora ; 1 rotate right Acc through carry
bcc bit0 ; 3 branch if carry is clear (is A&1=0)
mov #0x01,0x0000 ; 4 PTA0=hi
bra bit1      ; 3
bit0: mov #0x00,0x0000 ; 4 PTA0=lo
bit1: dbnzx loopsn ; 3   --> sum=11 or 14 cycles
```

```
fin: bsr delay
bsr delay
mov   #0x01,0x0000 ; PTA: PTA0 hi : STOP bit
bsr delay
bsr delay
pulx
rts

rcv: lda #0x80
rcvst: brset #0,*0x0000,rcvst; wait for START bit
; mov   #0x03,0x0003 ; START bit => switch LED off (debug)
bsr  delay ; wait half a bit width
looprc: bsr delay ; X
bsr delay ; X
brclr #0,*0x0000,rcv0 ; 5 branch if bit is clr => branch if 0, C=bit
rcv0: rora ; 1 after 8 ror, carry=1
; 2 uses of carry bit in these 2 lines: brclr sets the carry bit to the bit
; value, and we use the output of rora to the carry bit to check if we rotated
; 8 times already ...
bcc looprc ; 3   --> sum=8 or 11 cycles
bsr delay
bsr delay ; wait for stop bit
rts


delay: pshx ; 2+4 for bsr   (12+2*((X*9)+15))*.3333=833
ldx #0h0f ; 3                                    104
loopx: nsa ; 3    => Xinit=0x88 for 1200
nsa ; 3           =0x0f for 9600
dbnzx loopx ; 3
pulx ; 2
rts ; 4
```

`rs_rcsn.asm`: sample program for reading values on the RS232 port, adding 3 to them and sending them back.

```
#include "rs232.h"

void read_osc(int fd)
{unsigned char buf,cpt=10;
 while (1) {buf=cpt;write(fd,&buf,1);
            cpt++;
            printf ("%u($%x) -> ",buf&0x000000FF,buf&0x000000FF);
//            read(fd,&buf,1);
//            printf ("%u($%x) ",buf&0x000000FF,buf&0x000000FF);
            read(fd,&buf,1);
            printf ("%u($%x) \n",buf&0x000000FF,buf&0x000000FF);
            fflush(stdout);}
}

void main(int argc,char **argv)
{int fd;
 fd=init_rs232();
 read_osc(fd);
 free_rs232();
}
```

`hc08sndrec.c`: accompanying C program for testing `rs_rcsn`. This program sends values to the microcontroller and reads the processed result.

5 Storing a program in flash memory and execution

Flash memory starts at address 0xDC00. The limitation for the flash memory programmer is as follows:
it must be stored in RAM between locations 0x8C (since RAM space from 0x40 to 0x8B is used by the flash programmer provided in ROM of the 68HC08JB8) and 0xF9 (since that is the lowest address accessed by the stack in monitor mode). Hence, the flash programmer cannot be longer than 0x8C-0xF9=109 bytes.

Due to this limitation, we strip our flash programmer to the bare minimum: no character is sent from the microcontroller to the PC during the programming step (so the UART transmit emulation can be removed). This limitation removes the ability to echo the received bytes in order to confirm what is happening on the microcontroller side.

```
ldhx  #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov   #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f

mov   #0x00,0x0003 ; PTD0/1 lo => LED lit
mov   #0x03,0x0007 ; DDRD: PTD0/1 as output

ldhx #0xdc00
start: mov   #0x00,0x0004 ; DDRA: PTA0 as input
bsr rcv ; receive bit to be written in A
sta 0x100
bsr flash
lda 0,x
mov   #0x01,0x0000 ; PTA: PTA0 hi (in order to avoid glitches)
mov   #0x01,0x0004 ; DDRA: PTA0 as output
bsr  send
aix #1 ; H:X+=1
bra  start

flash: lda #0x01
sta 0xfe08 ; set PGM bit in FLCR
lda 0xfe09 ; read FLBPR
lda #0xff ; REQUIRED
sta 0xfe09 ; read FLBPR    ; REQUIRED
sta 0,x ; write to any area of row ***
lda #05
d5us1: dbnza d5us1 ; 3 cycles => 1 us/boucle: 5 us delay
lda #0x9
  sta 0xfe08 ; FLCR : PGM bit=1, HVEN=1
lda #0x0a
d5us2: dbnza d5us2 ; 3 cycles => 1 us/boucle: 10 us delay
lda 0x100 ; we have put the datum to be programmed on stack ***
sta 0,x ; ADDRESS TO BE WRITTEN ***
lda #0x14
d5us3: dbnza d5us3 ; 3 cycles => 1 us/boucle: 20 us delay
lda #0x08
  sta 0xfe08 ; FLCR : PGM bit=0
lda #0x05
d5us5: dbnza d5us5 ; 3 cycles => 1 us/boucle: 5 us delay
        lda #0x00
sta 0xfe08 ; FLCR : HVEN bit=0
lda #0x01
d5us6: dbnza d5us6 ; 3 cycles => 1 us/boucle: 1 us delay
rts
```

```
send: pshx
ldx #0x08 ;   snd through PTA0 the content of Acc (@9600)
mov   #0x00,0x0000 ;   PTA: PTA0 lo : START bit
loopsn: bsr delay ; X
bsr delay ; X
rora ; 1 rotate right Acc through carry
bcc bit0 ; 3 branch if carry is clear (is A&1=0)
mov #0x01,0x0000 ; 4 PTA0=hi
bra bit1      ; 3
bit0: mov #0x00,0x0000 ; 4 PTA0=lo
bit1: dbnzx loopsn ; 3   --> sum=11 or 14 cycles
fin: bsr delay
bsr delay
mov   #0x01,0x0000 ; PTA: PTA0 hi : STOP bit
bsr delay
bsr delay
pulx
rts

rcv: lda #0x80
rcvst: brset #0,*0x0000,rcvst; wait for START bit
; mov   #0x03,0x0003 ; START bit => switch LED off (debug)
bsr  delay ; wait half a bit width
looprc: bsr delay ; X
bsr delay ; X
brclr #0,*0x0000,rcv0 ; 5 branch if bit is clr => branch if 0, C=bit
rcv0: rora ; 1 after 8 ror, carry=1
; 2 uses of carry bit in these 2 lines: brclr sets the carry bit to the bit
; value, and we use the output of rora to the carry bit to check if we rotated
; 8 times already ...
bcc looprc ; 3   --> sum=8 or 11 cycles
bsr delay
bsr delay ; wait for stop bit
rts

delay: pshx ; 2+4 for bsr   (12+2*((X*9)+15))*.3333=833
ldx #0h0f ; 3                                    104
loopx: nsa ; 3    => Xinit=0x88 for 1200
nsa ; 3            =0x0f for 9600
dbnzx loopx ; 3
pulx ; 2
rts ; 4
```

`flash_write.asm`: program to be executed from the 68HC908JB8 RAM for reading values on PTA0 at 9600 bauds and store them in flash memory (starting at 0xDC00).

```
start: ldhx  #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov   #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f

mov   #0x00,0x0003 ; PTD0/1 lo => LED lit
mov   #0x03,0x0007 ; DDRD: PTD0/1 as output

mov   #0x01,0x0000 ; PTA: PTA0 hi (in order to avoid glitches)
mov   #0x01,0x0004 ; DDRA: PTA0 as output

lda #0hff
bcl: bsr delay
nsa
nsa
dbnza bcl

ldhx #0hdc00 ; STARTING ADDRESS TO BE READ (fc00 or dc00)

loop: lda 0h00,x
bsr  send ; send value of counter to serial port
aix #1 ; increment counter
cphx #0xffff
bne loop
end: bra  end

send: pshx
```

```
ldx #0x08 ;   snd through PTA0 the content of Acc (@9600)
mov   #0x00,0x0000 ;   PTA: PTA0 lo : START bit
loopsn: bsr delay ; X
bsr delay ; X
rora ; 1 rotate right Acc through carry
bcc bit0 ; 3 branch if carry is clear (is A&1=0)
mov #0x01,0x0000 ; 4 PTA0=hi
bra bit1      ; 3
bit0: mov #0x00,0x0000 ; 4 PTA0=lo
bit1: dbnzx loopsn ; 3   --> sum=11 or 14 cycles
fin: bsr delay
bsr delay
mov   #0x01,0x0000 ; PTA: PTA0 hi : STOP bit
bsr delay
bsr delay
pulx
rts

delay: pshx ; 2+4 for bsr   (12+2*((X*9)+15))*.3333=833
ldx #0h0f ; 3                                    104
loopx: nsa ; 3    => Xinit=0x88 for 1200
nsa ; 3            =0x0f for 9600
dbnzx loopx ; 3
pulx ; 2
rts ; 4
```

`flash_read.asm`: program to be executed from the 68HC908JB8 RAM for reading values in the flash memory (from 0xDC00 to 0xFFFF) and send them on PTA0 at 9600 bauds.

```
ldhx  #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov   #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f

mov   #0x00,0x0003 ; PTD0/1 lo => LED lit
mov   #0x03,0x0007 ; DDRD: PTD0/1 as output

ldhx #0xffe0
start: lda #0x06
sta 0xfe08 ; set ERASE and MASS bit in FLCR
lda 0xfe09 ; read FLBPR
lda #0xff ; REQUIRED
sta 0xfe09 ; read FLBPR    ; REQUIRED
sta 0,x ; write to any area of row ***
lda #05
d5us1: dbnza d5us1 ; 3 cycles => 1 us/boucle: 5 us delay
lda #0xe
  sta 0xfe08 ; FLCR : ERASE, MASS, HVEN=1

lda #0xfa
d5us2: dbnza d5us2 ; 3 cycles => 1 us/boucle: 250 us delay
lda #0xfa
d5us3: dbnza d5us3 ; 3 cycles => 1 us/boucle: 250 us delay
lda #0xfa
d5us4: dbnza d5us4 ; 3 cycles => 1 us/boucle: 250 us delay
```

```
lda #0xfa
d5us5: dbnza d5us5 ; 3 cycles => 1 us/boucle: 250 us delay
lda #0xfa
d5us6: dbnza d5us6 ; 3 cycles => 1 us/boucle: 250 us delay
lda #0xfa
d5us7: dbnza d5us7 ; 3 cycles => 1 us/boucle: 250 us delay
lda #0xfa
d5us8: dbnza d5us8 ; 3 cycles => 1 us/boucle: 250 us delay
lda #0xfa
d5us9: dbnza d5us9 ; 3 cycles => 1 us/boucle: 250 us delay
lda #0xfa
d5usa: dbnza d5usa ; 3 cycles => 1 us/boucle: 250 us delay

lda #0xc
  sta 0xfe08 ; FLCR : MASS, HVEN=1, ERASE=0

lda #100
d5usb: dbnza d5usb ; 3 cycles => 1 us/boucle: 100 us delay

lda #0x4
  sta 0xfe08 ; FLCR : MASS=1, HVEN=0, ERASE=0

mov   #0x03,0x0003 ; PTD0/1 lo => LED lit
end: bra end
```

`flash_erase.asm`: program to be executed from the 68HC908JB8 RAM for block erasing the whole flash memory (data and interrupt vector table).

Commands:
in order to erase the memory: `./hc08flash flash_erase.out`

in order to write the program `blink.out` to flash memory: `./hc08flash flash_write.out blink.out`
in order to check that the data were properly written: `./hc08flash flash_read.out > t`, which reads flash and ROM memories (from 0xDC00 to 0xFFFF) and stores the result in file `t` (for further verification of the memory content).

Of course these three steps can be automated by being combined in a shell script.

The programs `flash_write.asm` and `flash_erase.asm` do not use the subroutines provided in the ROM of the 68HC908JB8 since I was not able to figure out how to make them work.

6 USB communication

Now that we know how to store a program in flash memory, we can consider testing the sample software provided by Motorola with its evaluation board. The program is provided both in source format and compiled to an S19 format, `usb08.s19`. After converting this S19 file to a raw list of opcodes and data in hexadecimal format (by removing the S1 header and the address word at the beginning of each line, as well as the checksum byte at the end of each line), we store this program to flash memory by executing:
`hc08_flash flash_erase.out` in order to clear the flash memory
`hc08_flash flash_write.out usb08.flash` in order to write the new program to flash memory
`hc08_flash flash_read.out > usb08.dump` in order to read the content of the flash memory and check that the new program is indeed stored there
`hc08_flash flash_irq.out usb08.irq` in order to program the interrupt vector.

`flash_irq.asm`: program to be executed from the 68HC908JB8 RAM for programming the interrupt vector area (0xFFF0-0xFFFD).

For the last step, we have created a short file containing the hexadecimal values to be stored in the interrupt vectors located in 0xFFF0 to 0xFFFD. We must be careful *not* to write to 0xFFFE:FFFF or the microcontroller will no longer enter monitor mode when powered on (`flash_irq.asm` should never write to locations 0xFFFE:FFFF, even if the interrupt vector for this location is included in the file `usb08.irq`).

Since we now wrote in the interrupt vectors space, we must adapt the security bytes sent during initialization of the monitor mode (as done by the function `void init_hc08mon(int )` in the HC08 library `hc08.c`). We could until now always send the security bytes 0xFF since the interrupt vectors were never set. However, if we now try to read the content of flash memory using `hc08_flash flash_read.out`, we end up only reading the same value (0xAD in my case). We must provide the proper values for the security bytes in order to be able to read the content of flash memory and execute the program stored there. Indeed, by adapting the function `init_hc08mon` to send the same bytes as the ones read in `usb08.irq`, we can again read the content of the flash memory and thus execute it content (by running `hc08_flash` without argument, which will automatically call the program starting at 0xDC00, beginning address of the flash memory).

6.1 First USB tests under MS-Windows

Motorola provides with its USB development board a driver and software for controlling their microcontroller. This software requires a computer running MS Windows98 at least.

Running the demo version of USBIO provided by Thesycon while the board is connected through its USB link to the PC leads to an "USBIO Installation wizard" which properly lists our microcontroller as
Manufacturer: Thesycon
Descritpion: USBIO Device: VID=0C70 PID=0000 Hardware ID: USB
VID_0C70&PID_0000&REV_0100
after using the driver found in `usbio_lt.sys`.

In the Control Panel/System Properties one can indeed see the USBIO Device with the same VID and PID appear each time the board is plugged in the USB port (while the Motorola software is running on the microcontroller).

However, when running the demo application provided by Motorola (`io08usb.exe`) I get the error message "Couldn't open port! Please restart application to retry!". The computer I tried this on was equipped with an Intel 82801AA USB Universal Host Controller.

6.2 USB development under linux

Our objective was to provide a linux device driver and a user mode application running directly with the microcontroller USB example provided by MCT. We used the free evaluation version of the ICC compiler under Windows98 to generate the S19 file out of the C source code. The resulting S19 file was flashed in the HC908JB8 using the linux programming code following the procedure presented earlier in this document.

6.2.1 The linux (kernel 2.4.x) driver

`hc08.c`: linux kernel 2.4.x driver for communicating with the 68HC908JB8 microcontroller running the example C program provided by MCT.

6.2.2 The user client

`start_hc08`: shell script for loading the module and generating the (two) appopriate `/dev` entries for communicating with the microcontroller (up to two microcontrollers supported). This script is not to be run if `devfs` is used (ie the `devfsd` daemon is running).

`hc08_user_write`: user programm for writing values to the microcontroller. The LED (connected to PTA0/1) should light if 0xff is sent, and should switch off is 0x00 is sent.

`hc08_user_read.c`: user programm for writing values to the microcontroller and at the same time displaying the status of keys connected to PTE.