C51 In-System FLASH Programming

1. Introduction

Most C51 microcontroller based applications using an on-board FLASH memory for code and data storage take the advantage of being In-System Programmabled thanks to the FLASH technology. This application note describes the basic hardware and software requirements to build such a system based on a C51 product (or C251 product configured for C51 memory model). Basic In-System Programming features described in this document are: FLASH identifying, erasing and programming from a HEX file.

The C51/C251 Demo Board demonstrates this In-System Programming implementation using an Atmel AT49HF010 FLASH memory.

2. Description

2.1. System Requirements

The In-System FLASH Programming (ISP) described in this document assumes a typical C51 configuration shown in Figure 1.

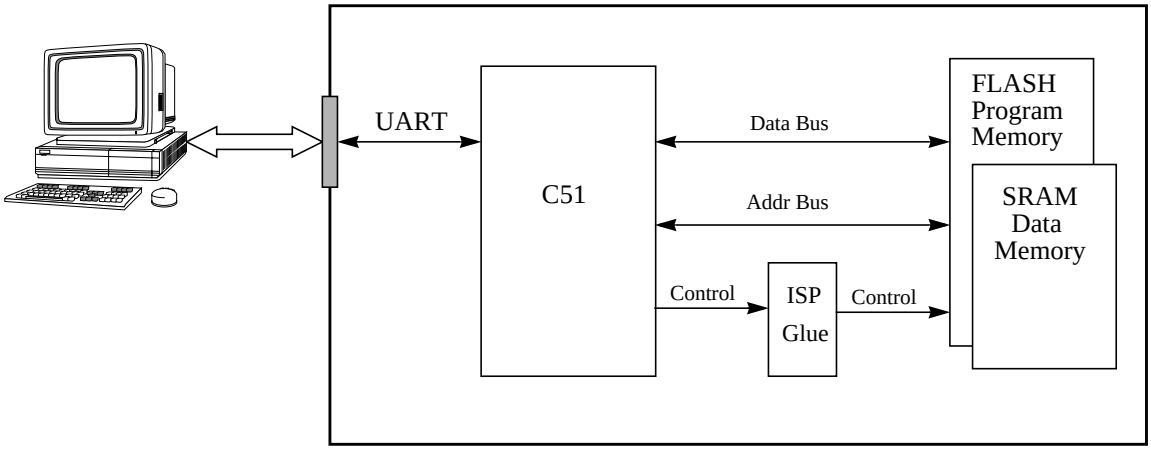

Figure 1. Typical C51 Configuration

The basic system includes a C51, a FLASH memory for code and a SRAM for data. The C51 interfaces the memory devices through some ISP dedicated glue to allow the C51 to write into the FLASH as if it would be a SRAM. The C51 is also connected to a terminal via a RS-232 serial link using its embedded UART. This terminal will be used to download a new program into the FLASH memory.

2.2. Memory Mapping

C51 microcontrollers have several memory space areas:

- The internal data memory
- The internal Special Function Register (SFR) memory to address on-chip resources
- The optional expanded on-chip data memory (on-chip XRAM)
- The external program memory
- The external data memory

2.2.1. Standard Program Execution Mode

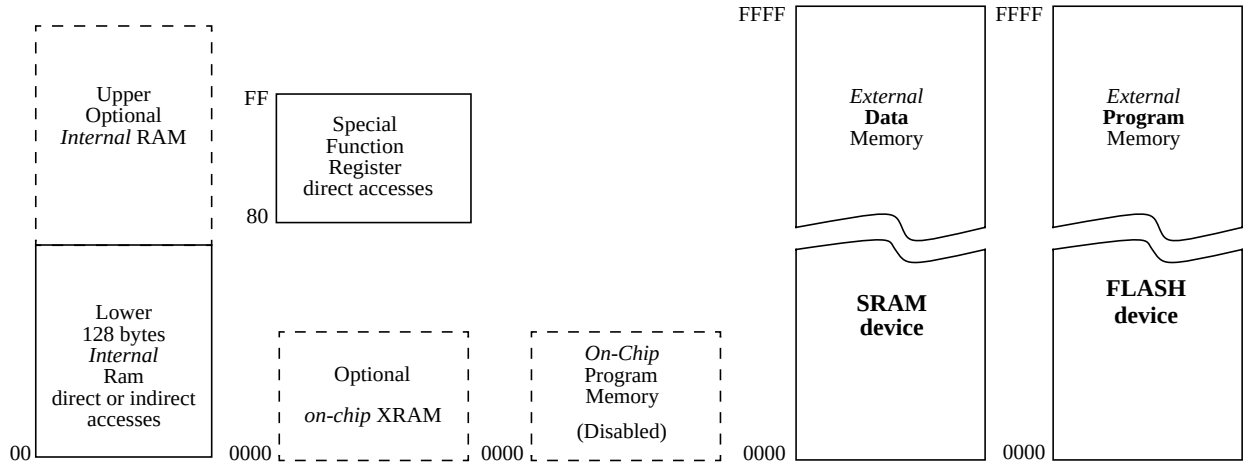

Figure 2. C51 Memory Mapping in Standard Mode

When the application is running (standard mode), the SRAM is used for application data storage and the FLASH memory for code and constant data storage. Data in SRAM are accessed thanks to the MOVX instruction and constant data are retrieved from the FLASH memory thanks to the MOVC.

2.2.2. ISP Program Execution Mode

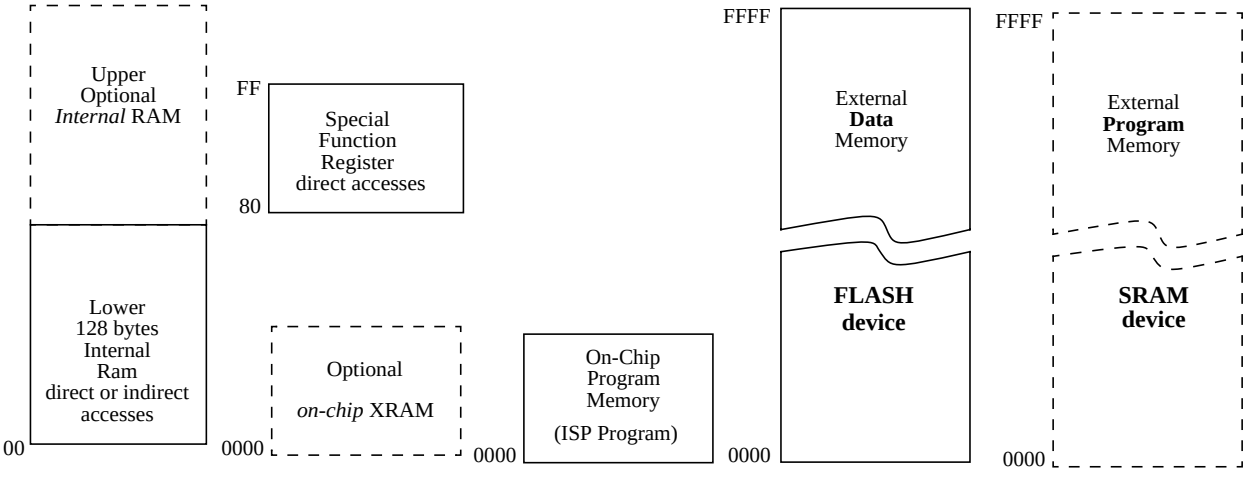

Figure 3. C51 Memory Mapping in ISP Mode

In ISP mode, the FLASH memory is re-mapped in the data space area so that a new program can be writen using data write instructions (MOVX). In this mode, the FLASH memory is no longer available for program execution and the ISP program must carried out from the on-chip program memory. Because the C51 microcontroller has a Harward architecture (separate memory data/code address spaces) the FLASH memory can be easily re-mapped into the data space providing a simple logic glue and few control signals.

Some C51 derivatives provide an extra on-chip data memory area called on-chip XRAM. This area is overlaid with the external data memory accessed using MOVX instructions. In ISP mode, this on-chip XRAM must be disabled otherwise MOVX instruction would not access to a part of the FLASH memory but to the on-chip XRAM area.

2.2.3. Special Considerations

In order to simplify the discussion, it is assumed there is no need to store application routines in the on-chip memory and the C51 would behave as a ROMless version:

- In standard mode, the microcontroller would reset with EA#=0 and the application would start from address 0000h located in FLASH memory
- In ISP mode, the microcontroller would reset with EA#=1 and the ISP program would start from address 0000h in the on-chip ROM/EPROM or FLASH memory depending on the part.

However some applications may require to store some extra routines in the on-chip memory for several reasons like cost reduction using the same ROM version on many different applications, crypted ROM routines for secured applications, auto-configuring application with on-chip FLASH program memory... In this latter case, some room must be left for the ISP subroutines and the microcontroller would always reset with EA#=1. To help the reset service routine to determine whether the application is in standard mode or ISP mode, an extra signal should be provided by the application. Note that a special care should be taken for the interrupt routine design since some interrupts vectors may be shared between the ISP program and the application.

3. Hardware

Figure 4 shows the typical hardware configuration for an application providing the In-System FLASH Programming feature. The $\overline{ISP}$ input signal configures the system in Standard Mode or in In-System Programming Mode.

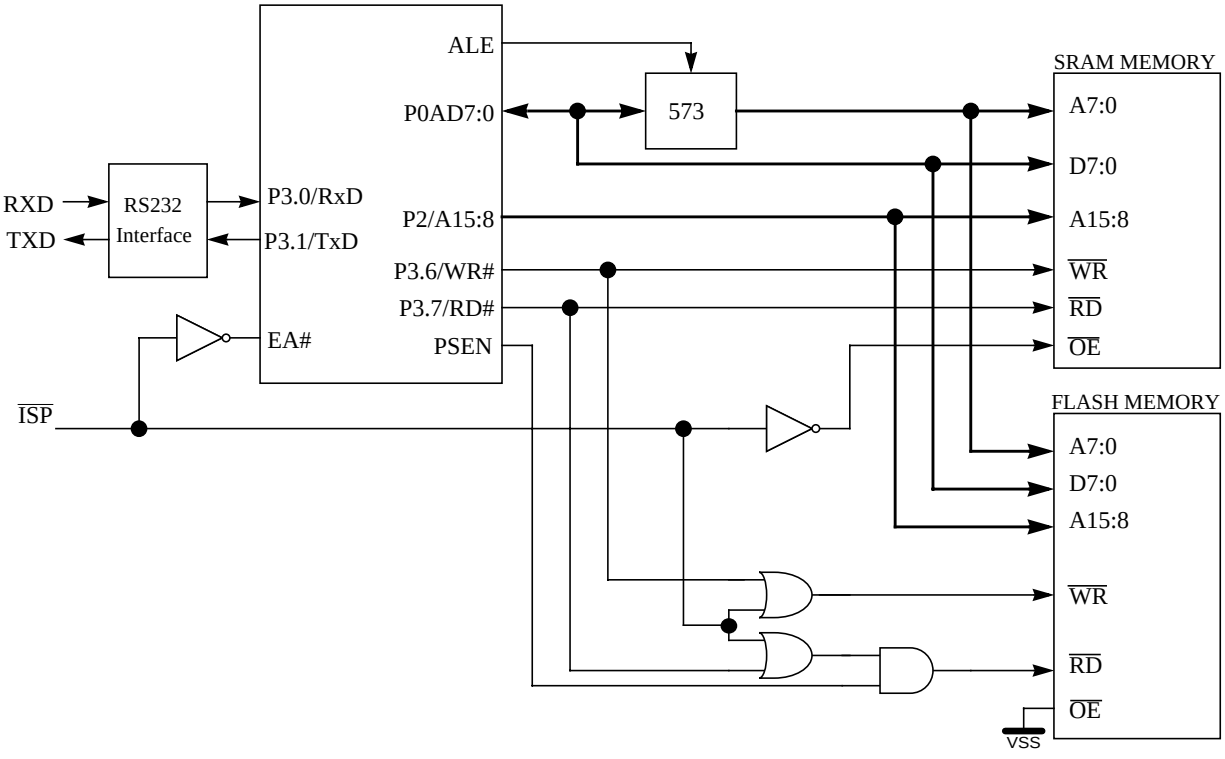

Figure 4. In-System Programming Hardware Configuration

When $\overline{ISP}$ is high, the standard mode is selected. After reset, the program branches at address 0000h which is located in the on-board FLASH memory (EA#=0).

When $\overline{ISP}$ is low, the ISP mode is selected. After reset, the program branches to address 0000h located in the on-chip memory and executes the ISP program.

The $\overline{ISP}$ also controls the FLASH / SRAM memory address area swap. The decoding equations for memory control signals are shown in Table 1. The glue logic that controls the memory signals may be programmed in a PLD (Programmable Logic Device).

Table 1. Equations for Memory Control

Operating Mode	FLASH Memory	SRAM Memory
$\overline{ISP}$ = 0 (In-System Programming Mode)	- $\overline{RD}$ = $\overline{PSEN}$ & RD# - $\overline{WR}$ = $\overline{WR}$	- $\overline{RD}$ = RD# - $\overline{WR}$ = $\overline{WR}$ - $\overline{OE}$ = 1
$\overline{ISP}$ = 1 (Standard Mode)	- $\overline{RD}$ = PSEN - $\overline{WR}$ = 1	- $\overline{RD}$ = X - $\overline{WR}$ = X - $\overline{OE}$ = 0

4. Software

The software in charge of the ISP is splitted into three parts: the IO interface, the Intel HEX file parser and the FLASH programming.

4.1. Operation

The ISP program performs the following tasks:

- Atmel FLASH authentification (AT49HF010)
- FLASH chip erase
- Download the HEX file and program data to the FLASH

For demonstration purpose, the host terminal displays the following messages during the ISP averall process:

```
** Welcome to the ISP program! **

Check if FLASH device is a Atmel AT49HF010... OK.

Erasing the on-board FLASH memory... OK.

Ready for FLASH programming.
Send .hex file with the following terminal configuration:
  - ASCII character transmission,
  - 8 bits, 1 stop, parity none,
  - XON-XOFF flow control.

Waiting for download...
.....................................................
.....................................................
.....................................
External FLASH memory is now programmed.
```

4.2. IO Interface

The HEX file is downloaded from a host via the standard C51 UART using a full-duplex communication protocol with a XON/XOFF data flow control. The program assumes a host as a simple terminal emulator that can display messages and download ASCII files.

The UART routines implement one circular buffer dedicated to the character reception. An interrupt service routine is in charge of the character reception so that FLASH programming can occur while receiving characters from the host.

- UART Reception

Upon reception of a character, the UART interrupt routine stores the received byte in the Rx circular buffer. As soon as the ISP program detects the Rx buffer is not empty, newly received characters are read from this buffer and are processed by the HEX file decoder. As soon as the Rx buffer is almost full, an XOFF character is automatically transmitted to indicate the terminal that reception shall stop. When a sufficient number of characters have been read and the Rx buffer is almost empty, an XON character is sent to the terminal to resume the transmission.

- UART Transmission

  Some simple messages are sent to the host for demonstration purpose. They are sent to the UART using the `printf` function provided by the Keil Development Kit [5]. This `printf` function is based on a `putchar` function that handles the XON/XOFF flow control.

4.3. Intel HEX File Parser

The Intel HEX file is a standard file format for binary program code encoding using ASCII characters (see [3]).

Basically, the Intel HEX file contains records. Each record is made of the following items:

- Record length: number of bytes in the record
- Record type: describes the information contained in the record (data, end of file, segment address, ...)
- Address field: indicated where the following bytes are stored in the memory
- Checksum

To minimize the data storage volume, the HEX file parser function analyses on a per byte basis each record. The HEX decoder decodes each record fields and extracts the address offset and the actual bytes that need to be programmed to the FLASH memory.

The parser structure is based on state machine shown in Figure 5.

Since the HEX file transmitted by the host encodes hexadecimal byte in ASCII form, the HEX decoder function must convert ASCII digits into binary format before processing. This conversion is made over 2 or 4 ASCII digits according to the record field (addresses are 4-digit numbers, data are 2-digit numbers).

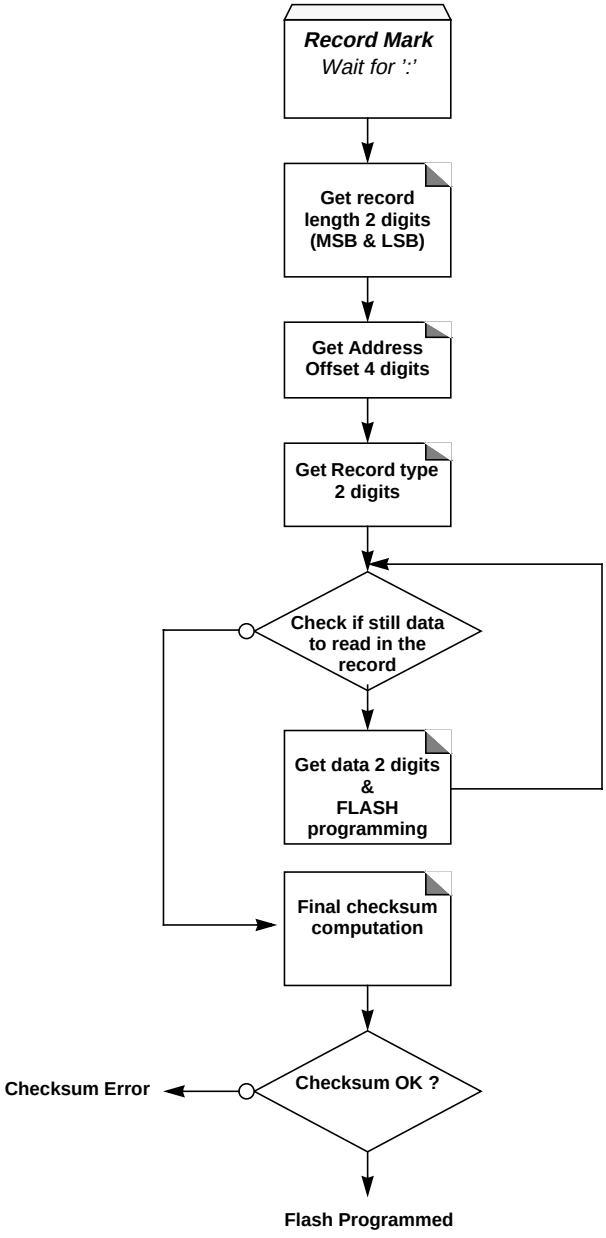

Figure 5. HEX File Decoder State Machine

4.4. FLASH Programming

A FLASH memory controller provides the user several write and erase operations like byte write, page write, page erase etc... The FLASH programming software routines implemented here provide the erasing of the whole FLASH memory array and the writing of one byte.

The FLASH memory controller handshaking required for an erase or write operation are detailled in [4]. The two basic sequences are shown Figure 6 and Figure 7.

4.4.1. Chip Erase

The Chip Erase command consists of a 6 cycle sequence. Upon reception of the command, the FLASH memory starts its erasure process. While erasure process is on-going, the bit 6 of the FLASH databus toggles each time a memory location is accessed. As soon as this bit stops flipping when accessed, the erasing operation is over.

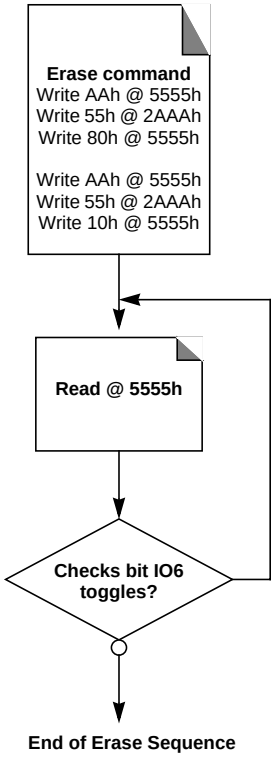

Figure 6. FLASH Erase Sequence

4.4.2. Byte Programming

The Byte Programming command consists of a 4 cycle sequence. Upon reception of the command, the FLASH memory waits for the data write sequence and starts its programming process. While programming process is on-going, the bit 7 of the FLASH databus is inverted each time the memory location is accessed. As soon as the correct data is read from the last programmed FLASH memory location, the programming operation is over.

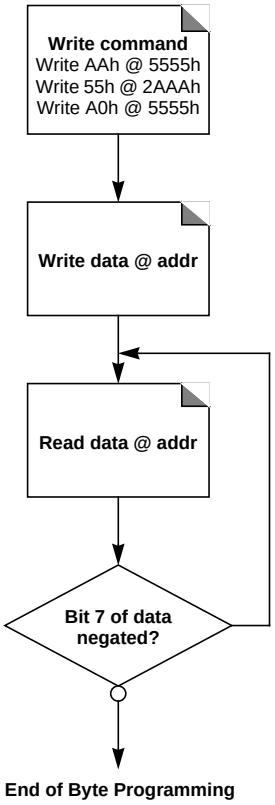

Figure 7. FLASH Write Sequence

5. Bibliography

[1] C51 Product Datasheets (Atmel Wireless & Microcontrollers)

[2] TSC80251G1 Design Guide ((Atmel Wireless & Microcontrollers)

[3] Hexdecimal Object File Format Specification (Intel)

[4] AT49HF010 FLASH Memory Datasheet (Atmel)

[5] Keil C51 C Compiler Manual

6. Sites to Visit

Atmel Wireless & Microcontrollers Web site: http://www.atmel-wm.com.com

7. Appendix A: Software

7.1. MAIN.C

```
/*C*****************************************************************************
* NAME: main.c
*-----------------------------------------------------------------------------
* PURPOSE:
* Main entry for ISP program and system initialization.
* Main entry includes all the ISP sequence from FLASH erase to FLASH prog.
*****************************************************************************/

/*_____ I N C L U D E S ____________________________________________________*/
#include <stdio.h>
#include "config.h"
#include "isp.h"

/*F*****************************************************************************
* NAME: _DEAD_
*-----------------------------------------------------------------------------
* PURPOSE:
* When a fatal error occurs, this function is called to end with a standard
* error message and a never end loop.
*****************************************************************************/
void _DEAD_(void)
{
  printf("\nError: cannot continue!\n");
  while (1);
}

/*F*****************************************************************************
* NAME: system_init
*-----------------------------------------------------------------------------
* PURPOSE:
* Initializes the microcontroller: XRAM disabled to enable the FLASH mapping
* into the data space and uart settings.
*****************************************************************************/
void system_init(void)
{
  /* Disable on-chip XRAM if exists */
#ifdef HAVE_XRAM
  AUXR = NO_XDATA;
#endif

  /* Initialize communication port */
  uart_init();
  EA = 1;
}

/*F*****************************************************************************
* NAME: main
*-----------------------------------------------------------------------------
* PURPOSE:
* Program main entry. Manages the basic ISP sequence flow from FLASH ersure
* to FLASH programming.
*****************************************************************************/
void main(void)
{
  Uchar status;
```

```
  /* System initialization: IO, welcome message, ... */
  system_init();
  printf("\n\n** Welcome to the ISP program! **\n\n");

  /* Check the Flash manufacturer and device Id (Atmel=1F, 49HF010=17) */
  printf("Check if FLASH device is a Atmel AT49HF010... ");
  if (flash_id()==0x1F17)
  {
    printf("OK.\n");
  }
  else
  {
    printf("KO!\n");
    _DEAD_();
  }

  /* On-board FLASH erase operation */
  printf("\nErasing the on-board FLASH memory... ");
  flash_erase();
  printf("OK.\n\n");

  /* FLASH erased: ready for programming */
  printf("Ready for FLASH programming.\n");
  printf("Send .hex file with the following terminal configuration:\n");
  printf("  - ASCII character transmission,\n");
  printf("  - 8 bits, 1 stop, parity none,\n"),
  printf("  - XON-XOFF flow control.\n");
  printf("\nWaiting for download...\n");

  /* Parses the downloaded HEX file and program the on-board FLASH memory */
  uart_rx_enable();

  status = hex_parser();

  uart_rx_disable();

  if (status == HEX_DEC_CSERR)
  {
    printf("\nChecksum error: external FLASH memory is not programmed!\n");
    _DEAD_();
  }
  else
    printf("\nExternal FLASH memory is now programmed.\n");

  /* Happy end! */
  while(1);
}
```

7.2. IO.C

```
/*C*****************************************************************************
* NAME: io.c
*-----------------------------------------------------------------------------
* PURPOSE:
* Functions dedicated to the HEX file reception from the host via the Uart.
* These functions provide uart initialization, interrupt handling and
* Rx circular buffer management.
*****************************************************************************/

/*_____ I N C L U D E S ____________________________________________________*/
#include "compiler.h"
#include "config.h"
#include "isp.h"

/*_____ M A C R O S ________________________________________________________*/
/* XON / XOFF Contro characters */
#define XON        17
#define XOFF       19

/* Must be 2^y for modulo computation */
#define RX_BUF_SIZE 16

/* XOFF when only 50% buffer size left */
/* XON when 25% rx buffer full */
#define XOFF_THRESH (RX_BUF_SIZE - (RX_BUF_SIZE / 2) )
#define XON_THRESH  (RX_BUF_SIZE / 4)

/* Wait for end of Tx over the Uart */
#define WAIT_EO_TX  {while (TI==0); TI=0;}

/*_____ D E F I N I T I O N ________________________________________________*/
static Uchar rx_buffer[RX_BUF_SIZE]; /* Rx circular buffer */
static Uchar rx_index_wr;            /* Rx circular buffer indexes */
static Uchar rx_index_rd;
static Bool  tx_off;                 /* XOFF state indicator */
Uchar nb_rx_data;                    /* Number of data in the Rx buffer */

/*_____ D E C L A R A T I O N ______________________________________________*/
void  rx_buffer_wr(Uchar rx_data);

/*F*****************************************************************************
* NAME: uart
*-----------------------------------------------------------------------------
* PURPOSE:
* Uart interrupt handler: processes Rx uart events only.
*****************************************************************************/
Interrupt(void uart(void),4)
{
  if (RI == 1)             /* Processes Rx event only, not Tx */
  {
    rx_buffer_wr(SBUF);    /* Writes the received data into the Rx buffer */
    RI = 0;
  }

}

/*F*****************************************************************************
```

```
* NAME: uart_tx
*-------------------------------------------------------------------------
* PURPOSE:
* Send a character over the serial link. Exit when the character
* transmission is over.
*************************************************************************/
void uart_tx(Uchar tx_data)
{
        SBUF = tx_data;
  WAIT_EO_TX;
}

/*F**************************************************************************
* NAME: uart_init
*-------------------------------------------------------------------------
* PURPOSE:
* Set the C51 Uart in 8-bit data, 9600 bauds, no parity operating mode.
*************************************************************************/
void uart_init(void)
{
  SCON = 0x50;
  TMOD = TMOD | 0x20 ;      /*  Timer1 in mode 2 & not gated */
  TH1 = 0xFD;               /*  9600 bauds at 11.059200 MHZ */
  TL1 = 0xFD;
  PCON = PCON & 0X80;
  TCON |= 0x40;
  TI=1;
}

/*F**************************************************************************
* NAME: uart_rx_enable
*-------------------------------------------------------------------------
* PURPOSE:
* Initializes Uart for data reception: circular buffer reset, XON/XOFF
* protocol initialization and interrupt enabled.
*************************************************************************/
void uart_rx_enable(void)
{
  rx_index_wr = rx_index_rd = nb_rx_data = 0;
  WAIT_EO_TX;                   /* Wait any previous transmission (printf) */
        uart_tx(XON);
  ES = 1;
}



/*F**************************************************************************
* NAME: uart_rx_disable
*-------------------------------------------------------------------------
* PURPOSE:
* Disable Rx reception upon interrupts.
*************************************************************************/
void uart_rx_disable(void)
{
  ES = 0;
  TI = 1;                       /* Enable Tx for printf (polling) */
}

/*F**************************************************************************
```

```
* NAME: rx_buffer_empty
*-------------------------------------------------------------------------
* PARAMS:
* return: TRUE when Rx buffer is empty.
*-------------------------------------------------------------------------
* PURPOSE:
* Check if new Rx data have been stored in the Rx buffer.
*************************************************************************/
Bool rx_buffer_empty(void)
{
  if (nb_rx_data == 0)
    return TRUE;
  else
    return FALSE;
}

/*F*************************************************************************
* NAME: rx_buffer_wr
*-------------------------------------------------------------------------
* PARAMS:
* rx_data: Rx data to store in the buffer.
*-------------------------------------------------------------------------
* PURPOSE:
* Stores the newly received data in the buffer and keep indexes updated.
*************************************************************************/
void rx_buffer_wr(Uchar rx_data)
{
  nb_rx_data++;
  rx_buffer[rx_index_wr] = rx_data;

  /* Circular buffer index computation */
  rx_index_wr = (rx_index_wr + 1) % RX_BUF_SIZE;

  /* Stops host transmission when more than XOFF_THRES characters are stored
   * in the Rx buffer */
  if ((tx_off==FALSE) && (nb_rx_data > XOFF_THRESH))
  {
               uart_tx(XOFF);
    tx_off = TRUE;
  }
}

/*F*************************************************************************
* NAME: rx_buffer_rd
*-------------------------------------------------------------------------
* PARAMS:
* return: the next data available from the Rx buffer.
*-------------------------------------------------------------------------
* PURPOSE:
* Retrieve the next data from the RX buffer.
*************************************************************************
* NOTE:
* All interrupts are disabled when updating rx_data because a Rx interrupt
* may occur and disturb the computation (rx_data also updated in the Uart
* interrupt handler.
*************************************************************************/
Uchar rx_buffer_rd(void)
{
        static Uchar data_cnt=0;                        /* For progression dots */
```

```
  Uchar rx_data;

  EA=0;                          /* Avoid conflicts with rx_buffer_wr() */
  nb_rx_data--;
  EA=1;

  rx_data = rx_buffer[rx_index_rd];

  /* Circular buffer index computation */
  rx_index_rd = (rx_index_rd + 1) % RX_BUF_SIZE;

  /* Progression dots every 256 data */
        data_cnt++;
        if (data_cnt == 0)
                uart_tx('.');

  /* Resumes host transmission when less than XON_THRES characters are stored
   * in the Rx buffer */
        EA = 0;
  if ((tx_off == TRUE) && (nb_rx_data < XON_THRESH))
  {
                uart_tx(XON);
    tx_off = FALSE;
  }
        EA = 1;

  return(rx_data);
}
```

7.3. FLASH.C

```
/*C*************************************************************************
* NAME: flash.c
*------------------------------------------------------------------------
* PURPOSE:
*************************************************************************/

/*_____ I N C L U D E S _____________________________________________________*/
#include "config.h"
#include "isp.h"

/*F*************************************************************************
* NAME: flash_wr
*------------------------------------------------------------------------
* PARAMS:
* addr: FLASH address location mapped in the external data area
* val: Data value to write to the FLASH.
*------------------------------------------------------------------------
* PURPOSE:
* Writes a byte to the FLASH memory when located in the external data area.
*************************************************************************/
void flash_wr(Uchar xdata *addr, Uchar val)
{
  *addr = val;          /* addr is a pointer to external data mem */
}

/*F*************************************************************************
* NAME: flash_rd
*------------------------------------------------------------------------
* PARAMS:
* addr: FLASH address location mapped in the external data area
* return: Byte value read from the FLASH memory
*------------------------------------------------------------------------
* PURPOSE:
* Reads a byte from the FLASH memory when located in the external data area.
*************************************************************************/
Uchar flash_rd(Uchar xdata *addr)
{
  return *addr;
}

/*F*************************************************************************
* NAME: flash_cmd
*------------------------------------------------------------------------
* PARAMS:
* cmd: FLASH command used by the FLASH command sequence
*------------------------------------------------------------------------
* PURPOSE:
* Performs a FLASH command sequence (on-FLASH memory controller configuration)
* The command is defined by the cmd code (erase, chip-id access, ...)
*************************************************************************/
void flash_cmd(Uchar cmd)
{
  flash_wr(0x5555, 0xAA);
  flash_wr(0x2AAA, 0x55);
  flash_wr(0x5555, cmd);
}
```

```
/*F*****************************************************************************
* NAME: flash_erase
*------------------------------------------------------------------------------
* PURPOSE:
* Erases the entire FLASH memory.
*****************************************************************************/
void flash_erase(void)
{
  Uchar pol_n, pol_n_1;

  /* Erase command sequence */
  flash_cmd(0x80);
  flash_cmd(0x10);

  /* Toggle bit algorithm: IO6 toggles each time a data read occurs */
  /* End of toggoling signals end of erase */
  pol_n_1 = flash_rd(0x5555);
  pol_n   = flash_rd(0x5555);

  while ((pol_n ^ pol_n_1) == 0x40) /* Checks if bit6 has changed between
                                       2 polls */
  {
    pol_n_1 = pol_n;
    pol_n   = flash_rd(0x5555);
  }
}

/*F*****************************************************************************
* NAME: flash_prog
*------------------------------------------------------------------------------
* PARAMS:
* addr: FLASH address location mapped in the external data area
* val: Data value to program to the FLASH.
*------------------------------------------------------------------------------
* PURPOSE:
* Programs one byte to the FLASH memory.
*****************************************************************************/
void flash_prog(Uint16 addr, Uchar value)
{
  /* Programming command */
  flash_cmd(0xA0);

  flash_wr(addr, value);

  /* Wait until end of programming: IO7 is negated until end of programming */
  while (flash_rd(addr) != value);
}

/*F*****************************************************************************
* NAME: flash_id
*------------------------------------------------------------------------------
* PARAMS:
* return: FLASH memory manufacturer id and device id bytes.
*         MSB is Manufacturer Id and LSB Device Id.
*------------------------------------------------------------------------------
* PURPOSE:
* Read the manufactuer and device Id's.
*****************************************************************************/
Uint16 flash_id()
```

```
{
  Uint16 flash_id=0;

  /* Product Id Entry mode */
  flash_cmd(0x90);

  /* @0000: manufacturer, @0001: device */
  flash_id = flash_rd(0x0000) << 8 | flash_rd(0x0001);

  /* Exit from Product Id */
  flash_cmd(0xF0);

  return flash_id;
}
```

7.4. HEX.C

```
/*C*****************************************************************************
* NAME: hex.c
*-----------------------------------------------------------------------------
* PURPOSE:
* HEX file decoder: extracts HEX file information, retrieve data and program
* the FLASH memory on the fly.
*****************************************************************************/

/*_____ I N C L U D E S ____________________________________________________*/
#include "config.h"
#include "isp.h"
#include <ctype.h>

/*_____ M A C R O S ________________________________________________________*/
/* State definition of the HEX file decoder state machine */
#define REC_MARK    0x01
#define REC_LEN_1   0x02
#define REC_LEN_2   0x03
#define OFFSET_1    0x04
#define OFFSET_2    0x05
#define OFFSET_3    0x06
#define OFFSET_4    0x07
#define REC_TYP_1   0x08
#define REC_TYP_2   0x09
#define DATA_1      0x0A
#define DATA_2      0x0B
#define CHEKSUM_1   0x0C
#define CHEKSUM_2   0x0D

/*_____ D E C L A R A T I O N ______________________________________________*/
Uchar hex_decoder(Uchar hex_data);

/*F*****************************************************************************
* NAME: hex_parser
*-----------------------------------------------------------------------------
* PARAMS:
* return: Exit status of the HEX file parser: OK or bad CRC encountered.
*-----------------------------------------------------------------------------
* PURPOSE:
* Monitors the HEX file Rx buffer and call the HEX decoder as soon as data
* are available.
*****************************************************************************/
Uchar hex_parser(void)
{
  Uchar hex_data, status;

  status = HEX_DEC_OK;

  while (status == HEX_DEC_OK)  /* Processes while hex decoder status is OK */
  {
    if(nb_rx_data != 0)
    {
      hex_data = rx_buffer_rd();
      status = hex_decoder(hex_data);
    }
  }
  return status;
```

```
}

/*F****************************************************************************
* NAME: hex_decoder
*-----------------------------------------------------------------------------
* PARAMS:
* hex_data: data from the HEX file to decode
* return: Hex decoder exit status: OK or bad CRC encountered.
*-----------------------------------------------------------------------------
* PURPOSE:
* Decodes all HEX file records on a byte per byte basis: analyse HEX record,
* extracts data for FLASH programming and verifies the checksum.
*****************************************************************************
* NOTE:
* HEX data are ASCII data format. Each byte is made of 2 ASCII form digits
* that needed to be converted. Conversion uses toint which is not a ANSI
* C function. One should use strtol when available (not in Keil C lib).
*****************************************************************************/
Uchar hex_decoder(Uchar hex_data)
{
  static state = REC_MARK;
  static Uchar length, type, nb_byte, data_value, sum, sum_1, sum_2;
  static Uint16 offset;

  Uchar status;

  status = HEX_DEC_OK;

  switch(state)
  {
    case REC_MARK:            /* Start of a new record */
          {
      if (hex_data == ':')    /* Check if the right character */
        {
          state = REC_LEN_1;  /* If ok, next step: get the 1st lenght char */
          nb_byte=0;
        }
        else
          state = REC_MARK;   /* If ko, spurious char and skip */
      break;
    }

    case REC_LEN_1:           /* Get the 1st digit of the length byte */
    {
      length = toint(hex_data) * 16;
            state = REC_LEN_2;
      break;
    }

    case REC_LEN_2:           /* Get the 2nd digit of the lenght byte */
    {
            length = length + toint(hex_data);
      state = OFFSET_1;
      break;
    }

    case OFFSET_1:            /* Get the 1st digit of the adress byte */
    {
            offset = toint(hex_data) * 4096;
```

```
  state = OFFSET_2;
  break;
}

case OFFSET_2:            /* Get the 2nd digit of the adress byte */
{
        offset = offset + toint(hex_data) * 256;
    sum_1= offset / 256;
        state = OFFSET_3;
  break;
}

case OFFSET_3:            /* Get the 3rd digit of the adress byte */
{
  offset = offset + toint(hex_data) * 16;
        state = OFFSET_4;
  break;
}

case OFFSET_4:            /* Get the 4th digit of the adress byte */
{
         offset = offset + toint(hex_data);
         sum_2 = offset - sum_1 * 256;
   state = REC_TYP_1;
   break;
}

case REC_TYP_1:           /* Get the 1st digit of the record type */
{
   type = toint(hex_data) * 16;
         state =REC_TYP_2;
   break;
}

case REC_TYP_2:           /* Get the 2nd digit of the record type */
{
  type = type + toint(hex_data);
  sum = length + sum_1 + sum_2 + type;
  if (length==0x00)
    state=CHEKSUM_1;      /* If no data, go to checksum computation */
  else
    state = DATA_1;       /* Otherwise next is data record acquisition */
  break;
}

case DATA_1:              /* Get the 1st digit of one data */
{
  data_value = toint(hex_data) * 16 ;
        state = DATA_2;
  break;
}

case DATA_2:              /* Get the 2nd digit of one data */
{
        data_value = data_value + toint(hex_data);
        sum = sum + data_value;

  flash_prog(offset + nb_byte, data_value);
```

```
      nb_byte++;
      if (nb_byte == length)
       state = CHEKSUM_1;     /* If end of data block, go to checksum */
      else
       state = DATA_1;        /* Otherwise, get the next data in the block */
      break;
     }

     case CHEKSUM_1:          /* Get 1st digit of the checksum */
     {
       sum_1 = toint(hex_data) * 16;
             sum = (~sum) + 1;      /* Checksum 2 complement */
       state = CHEKSUM_2;
       break;
     }

     case CHEKSUM_2:          /* Get 2nd digit of the checksum */
     {
       sum_1 = sum_1 + toint(hex_data);
       if (sum_1 != sum)
         status = HEX_DEC_CSERR;

       state = REC_MARK;      /* Ready for next record */
       if(type==0x01)
         status = HEX_DEC_END; /*  end of transmission */
       break;
    }
  }
  return status;
}
```

7.5. CONFIG.H

```
/*H*************************************************************************
* NAME: myfile.h
*-------------------------------------------------------------------------
* PURPOSE:
*************************************************************************/

#ifndef _CONFIG_H_
#define _CONFIG_H_

/*_____ I N C L U D E S ____________________________________________________*/
#include <reg51.h>
#include "compiler.h"

/* ___________________________ User Configuration Section ___________________________ */

#define HAVE_XRAM              /* Target chip: T89C51RD2 with on-chip XRAM */

/* ___________________________ User Configuration Section ___________________________ */


#ifdef HAVE_XRAM               /* If on-chip XRAM, disable the XRAM to access to the
entire */
#define NO_XDATA 0x02          /* FLASH when mapped in the data area */
Sfr (AUXR, 0x8E);
#endif

#endif /* _CONFIG_H_ */
```

7.6. COMPILER.H

```
/*H*****************************************************************************
* NAME: compiler.h
*-----------------------------------------------------------------------------
* PURPOSE:
* Defines compiler dependant definitions to enhance program portability.
* Allows to remap exotic syntaxes to another exotic syntaxe without changing
* C source files.
*****************************************************************************/

#ifndef _COMPILER_H_
#define _COMPILER_H_

/*_____ M A C R O S ________________________________________________________*/
#define FALSE  0
#define TRUE   1

/*_____ D E F I N I T I O N ________________________________________________*/
typedef unsigned char   Uchar;
typedef unsigned short  Uint16;
typedef signed int      Int16;
typedef float           Float16;
typedef unsigned char   Bool;

/* KEIL compiler syntax redefinition */
#define Reentrant(x)    x reentrant
#define Sfr(x,y)        sfr x = y
#define Sbit(x,y,z)     sbit x = y ^ z
#define Interrupt(x,y)  x interrupt y
#define At(x)           _at_ x

#endif /* _COMPILER_H_ */
```

7.7. ISP.H

```
/*H**************************************************************************
* NAME: isp.h
*-------------------------------------------------------------------------
* PURPOSE: Header file shared by all ISP C files.
**************************************************************************/

#ifndef _ISP_H_
#define _ISP_H_

/* Uart */
extern Uchar nb_rx_data;
void  uart_init(void);
void  uart_rx_enable(void);
void  uart_rx_disable(void);
Bool  rx_buffer_empty(void);
Uchar rx_buffer_rd(void);

/* HEX file parser */
#define HEX_DEC_OK    0x01
#define HEX_DEC_END   0x02
#define HEX_DEC_CSERR 0x03
Uchar hex_parser (void);

/* FLASH API's*/
void flash_erase(void);
void flash_prog(Uint16, Uchar value);
Uint16 flash_id(void);

#endif /* _ISP_H_ */
```