



Chapter 9

FIPSOC Boot Program

FIPSOC
User's Manual

FIPSOC Boot Program

Overview

The Field Programmable System On Chip (FIPSOC) constitutes a new concept in system integration. It provides the user with the possibility of integrating a microprocessor core along with programmable digital and analog cells within the same integrated circuit. This chip can be considered as a large granularity FPGA with a FPAA (Field Programmable Analog Array) and a built-in microprocessor core that does not only act as a general purpose processing element, but also configures the programmable cells and their interconnections. Therefore, there is a strong interaction between hardware and software as long as signal values and configuration data within the programmable cells are accessible from microprocessor programs.

This chapter describes the program stored in internal ROM used to boot the device from a serial stream. The program interprets in real time an extension of Intel's HEX records. Commands are provided for programming internal (both directly and indirectly) and external RAM, reading internal (both directly and indirectly) and external RAM and code ROM, and program branching and returning. SFRs can then be accessed this way as long as direct writes to the internal memory can be done.

The extended HEX records are taken either from the RS-232 serial port (typically coming from a PC) or from a serial memory (typically flash or E²PROM) supporting SPI or 2-wire protocols. RS-232 and SPI modes also support reading, so the whole chip can be controlled (programmed and read) using a single serial link (typically from a PC).

1. Boot modes

FIPSOC provides four boot modes which are selected from external pins boot1 and boot2. The address port AD[15:8] is sampled during reset to provide extra configuration information. Table 1 shows the four boot modes.

boot1	boot0	Boot mode
0	0	2-wire
0	1	SPI
1	0	SCI
1	1	External parallel ROM

Table 1: FIPSOC boot modes

The last FIPSOC boot mode can be read with the special function register (SFR) SBCR located at address \$9D:

7	6	5	4	3	2	1	0
SCEN	SCSL	Master	SCIE	GOE	OD	IREN	BAUD

SCEN (bit 7) - Serial Communication Enable: The serial communication system, which can be configured either as 2-wire or SPI, is enabled if this bit is high. Its value upon reset is the inverted value of the external BOOT1 pin.

SCSL (bit 6) - Serial Communication Select: The serial communication system is configured as 2-wire if this bit is set, SPI otherwise. Its value upon reset is the inverted value of the external BOOT0 pin.

Master (bit 5) - Boot Master: This bit stores the serial communication mode (master or slave) upon reset. Its value is the inverted value of bit #15 of the address port, AD[15], and is latched during the reset sequence.

SCIE (bit 4) - Serial Communication Interrupt Enable: It enables the interrupt of the serial communication system. Its reset value is zero.

GOE (bit 3) - Global Output Enable: When reset, it disables every output of every DMC and places all the IO pads in input state. Its reset value is zero.

OD (bit 2) - Open Drain: The general purpose port GPORT pins behave as open drain when configured as outputs when this bit is set, as normal CMOS outputs otherwise. Its reset state depends on the boot mode: if booting from SPI (BOOT1=0 and BOOT0=1), then OD is initialized from bit #10 of the address port, AD[10]; if booting from 2-wire (BOOT1=BOOT0=0), OD is initialized to 1; otherwise, OD is initialized to zero (normal CMOS outputs), although all bits in the data direction register DDRP are initialized to 1 thus configuring all port pins as inputs.

IREN (bit 1) - Internal ROM Enable: When set, the internal ROM where the boot program is located is mapped at locations \$0000 to \$001FF. The reset state of this bit is the OR function of the external BOOT mode pins, that is, the internal boot ROM is enabled when booting from a serial link and disabled if booting from external parallel ROM.

Baud (bit 0) - SCI boot baud rate: This bit selects the initial baud rate of the SCI port when booting from it: if set, $f_{osc}/4992$ is selected (3205,1 baud if a typical 16 MHz xtal is used), $f_{osc}/1536$ otherwise (10416,6 baud if a typical 16 MHz xtal is used). Its reset state is latched during reset from bit #8 of the address port, AD[8].

Bits 7, 6, 4, 3, 2 and 1 keep their meaning after reset and are used to control their corresponding specific FIPSOC features at any time. In particular the SPI or 2-wire interface could be independently used regardless of the boot mode.

1.1. Booting from external parallel ROM

This mode is entered when external pins boot1 and boot0 are both tied to 1 during the reset sequence. In this mode the 8051 is booted from an external parallel ROM as the 8051 standard device, starting execution at position \$0000.

1.2. Booting from a serial link

FIPSOC boots from a serial link when external pins boot1 and boot0 are not both tied to 1 during the reset sequence. In these three modes a boot program stored in a 512-byte internal ROM located at address 0000H is executed upon reset. The program is essentially the same for the three modes except for the source the data is fetched from, which is configured after sampling external pins boot1 and boot0 upon reset: In mode 10 (boot1=1 and boot0=0), data is received from the serial port (typically from a PC); in modes 00 (boot1=0 and boot0=0) and 01 (boot1=0 and boot0=1), data is fetched from an 2-wire or SPI interface respectively (typically from a serial flash or EEPROM).

1.2.1. Extended HEX records

After initialization, the boot program keeps interpreting in real time configuration commands similar to Intel's HEX records. The syntax of these records is as follows:

```
: <number_of_bytes> <address_high>
<address_low> <record_type>
{<data_byte>} <checksum>
```

Each field in a record is a single byte, including a leading 3AH (the ASCII code of the colon ":").

The *number of bytes* field only refers to the *data bytes* represented in brackets in the syntax above. This

way, a zero number of bytes field is possible, and the *checksum* field will closely follow the *record type* field.

The *checksum* field is a byte such as the following formula is satisfied:

$$(number_of_bytes + address_high + address_low + record_type + \{ data_byte \}) \bmod 256 + checksum = 0$$

Note that the leading colon (ASCII 3AH) is not included in the checksum calculation.

Up to nine record types are currently supported. They are explained below:

Record type 00H: Write to external memory. The general syntax for this record is the following:

```
: <number_of_bytes> <add_H> <add_L> 00H
{<byte>} <checksum>
```

Interpretation of this command writes *number_of_bytes* bytes specified in the *byte* fields in sequential memory locations of the external data RAM memory starting from address $add_H * 256 + add_L$. The number of bytes that can be written is limited to 63, and an attempt to write more than that will in general cause a checksum error.

Record type 08H: Read from external memory. The general syntax for this record is the following:

```
:01H <add_H><add_L> 08H
<number_of_bytes> <checksum>
```

Interpretation of this command reads *number_of_bytes* bytes from sequential memory locations of the external data RAM memory starting from address $add_H * 256 + add_L$. Data is sent through the serial link sequentially as it is read. This record is not supported if 2-wire communication mode is selected.

Record type 0CH: Read from code memory. The general syntax for this record is the following:

```
:01H <add_H><add_L> 0CH
<number_of_bytes> <checksum>
```

Interpretation of this command reads *number_of_bytes* bytes from sequential memory locations of the external code (ROM) memory starting from address $add_H * 256 + add_L$. Data is sent through the serial link sequentially as it is read. This record is not supported if 2-wire communication mode is selected.

Record type 10H: Indirectly write to internal memory. The general syntax for this record is the following:

```
: <number_of_bytes> <add_H> <add_L> 10H
{<byte>} <checksum>
```

Interpretation of this command writes *number_of_bytes* bytes specified in the *byte* fields in

sequential memory locations of the internal data RAM memory starting from address *add_L* using indirect addressing (the *add_H* field is ignored but used in the checksum). This implies that writes to addresses ranging from 80H to FFH will be done on data memory rather than SFRs. The number of bytes that can be written is limited to 63, and an attempt to write more than that will in general cause a checksum error.

Record type 18H: Indirectly Read from internal memory. The general syntax for this record is the following:

```
:01H      <add_H><add_L>      18H
<number_of_bytes> <checksum>
```

Interpretation of this command reads *number_of_bytes* bytes from sequential memory locations of the internal data RAM memory starting from address *add_L* using indirect addressing (the *add_H* field is ignored but used in the checksum). This implies that reads from addresses ranging from 80H to FFH will be done on data memory rather than SFRs. Data is sent through the serial link sequentially as it is read. This record is not supported if 2-wire communication mode is selected.

Record type 20H: Directly write to internal memory. The general syntax for this record is the following:

```
: <number_of_bytes> <add_H> <add_L> 20H
{<byte>} <checksum>
```

Interpretation of this command writes *number_of_bytes* bytes specified in the *byte* fields in sequential memory locations of the internal data RAM memory starting from address *add_L* using direct addressing (the *add_H* field is ignored but used in the checksum). This implies that writes to addresses ranging from 80H to FFH will be done on SFRs rather than data memory. The number of bytes that can be written is limited to 63, and an attempt to write more than that will in general cause a checksum error.

Record type 28H: Directly Read from internal memory. The general syntax for this record is the following:

```
:01H      <add_H><add_L>      28H
<number_of_bytes> <checksum>
```

Interpretation of this command reads *number_of_bytes* bytes from sequential memory locations of the internal data RAM memory starting from address *add_L* using direct addressing (the *add_H* field is ignored but used in the checksum). This implies that reads from addresses ranging from 80H to FFH will be done on SFRs rather than data memory. Data is sent through the serial link sequentially as it is read. This record is not supported if 2-wire communication mode is selected.

Record type 40H: Jump to address. The general syntax for this record is the following:

```
:00H <add_H><add_L> 40H <checksum>
```

Interpretation of this command produces a direct jump to address $\text{add_H} * 256 + \text{add_L}$. Depending on the booting mode, parts of the serial link circuitry may be powered down prior to jumping.

Record type 48H: Return from subroutine. The general syntax for this record is the following:

```
:00H <add_H><add_L> 48H <checksum>
```

Interpretation of this command executes a "RET" instruction, which pops the address to jump to from the stack. It is used when the record parsing program is used as a subroutine from an user application program. fields *add_H* and *add_L* are ignored but used in the checksum.

1.2.2. Extended record types and errors

Record types 80H to FFH are available for command set extension. If such a record type is parsed the program jumps to address \$FF2B where an appropriate parsing routine should be found. Upon initialization a default error routine is written there.

This error routine is entered whenever a colon is not found at the beginning of an HEX record, a checksum is incorrect, a record type is not recognized or any other parsing problem is detected. This routine resets the GOE bit to place the whole programmable logic in an idle mode and then keeps branching to the same address.

1.2.3. Booting from SPI

When boot1 is 0 and boot0 is 1 during the reset sequence, the extended HEX records are fetched from the SPI serial link. The address port AD is also sampled upon reset and automatically configures several features of the SPI interface according to external user-defined hardware settings:

7	6	5	4	3	2	1	0
Master	CPOL	CPHA	CK1	CK0	OD		

Master (bit 7): FIPSOC will act as an SPI master if this bit is set, slave otherwise.

CPOL (bit 6): Selects the SPI clock polarity (refer to the SPI section of this user's manual).

CPHA (bit 5): Selects the SPI clock phase (refer to the SPI section of this user's manual).

CK1 and CK0 (bits 4 and 3): Select the SPI clock frequency as shown in table 2 (refer to the SPI section of this user's manual).

CK1	CK2	SPI clock frequency	frequency for $f_{osc}=16\text{MHz}$
0	0	$f_{osc}/8$	2 MHz
0	1	$f_{osc}/16$	1 MHz
1	0	$f_{osc}/32$	500 KHz
1	1	$f_{osc}/64$	250 KHz

Table 2: SPI clock frequencies upon reset

OD (bit 2): Bit lines will be open-drain if OD is set to 1, normal CMOS outputs otherwise.

Bits1 and 0: not sampled upon reset.

If configured as master, the initialization sequence goes as follows:

step 1: bit 0 of the general purpose register is driven low (and the corresponding bit of the data direction register is set to "output") to select a slave through the slave select line, typically the serial memory.

step 2: Command 03H is sent through the line, which is interpreted by a serial SPI memory as a "read" command.

step 3: Two zero bytes are sent through the line, which is interpreted by a serial SPI memory as the initial memory location to start reading from.

step 4: A dummy FFH byte is sent through the line, which provokes the first real data byte to be obtained from the serial memory.

After initialization, the boot program keeps reading the incoming bytes and interpreting extended HEX records as they are completed. Once a byte has been received, the same byte is echoed back through the line to trigger the next read in the memory.

When a JUMP instruction is found (record type 40H), bit 0 of the general purpose port is released and configured again as input, and the SPI communications system is switched off before actually executing the jump.

If configured as a slave the program does the same except for that no initialization is performed other than placing an initial FFH in the outgoing shift register. Bytes are echoed back as they arrive, which can be checked out by the master to validate transfers.

1.2.4. Booting from 2-wire

When boot1 and boot0 are both 0 during the reset sequence, the extended HEX records are fetched from the 2-wire serial link. The general purpose port GPORT is also sampled upon reset and automatically configures several features of the 2-wire interface according to external user-defined hardware settings:

7	6	5	4	3	2	1	0
Master	Major device address				Minor device address		

Master (bit 7): FIPSOC will act as a 2-wire master if this bit is set, slave otherwise.

Bits 6-0: These bits set the 7-bit logical device address. The four most significant ones are said to be the *major device address* which is only dependent on the nature of the device itself. The three least significant ones are the *minor device address* and are the ones that change among instances of the same device.

Bit 0 also configures the clock rate upon reset (refer to the 2-wire section of this user's manual).

GPORT[0]	2-wire clock frequency	frequency for $f_{osc}=16\text{MHz}$
0	$f_{osc}/16$	1 MHz
1	$f_{osc}/160$	100 KHz

Table 3: 2-wire clock frequencies upon reset

If configured as master, the initialization sequence goes as follows:

step 1: The device waits until the 2-wire line is free, and then it sends a START command to grab the line.

step 2: Once the line has been taken, a WRITE command is sent to a device with address 1010000. If no acknowledge is received, then FIPSOC sends a RESTART command without releasing the line and sends a WRITE command to the next device address, 1010001. It keeps restarting and trying until a device answers.

step 3: Once a device answers by acknowledging the ninth bit, two consecutive zeroes are sent to the line to specify the address to start reading from.

step 4: If no error is detected (otherwise a STOP command is issued and the program goes back to step 1), a RESTART command is issued and a READ command is sent to the device address that answered before.

step 5: A dummy FFH byte is sent through the line, which provokes the first real data byte to be obtained from the serial memory.

After initialization, the boot program keeps reading the incoming bytes and interpreting extended HEX records as they are completed. Once a byte has been received, the same byte is echoed back through the line to trigger the next read in the memory.

When a JUMP instruction is found (record type 40H), a STOP command is issued, and the 2-wire communications system is switched off before actually executing the jump. When a RETURN instruction is found (record type 48H), only the STOP command is issued before returning.

If configured as a slave, the program waits to be addressed (a WRITE command is received with the appropriate device address) and then sends a dummy FFH to acknowledge the reception. After that, the program keeps receiving bytes through the 2-wire line and interpreting extended HEX records as they

are completed. When a JUMP instruction is found (record type 40H), the 2-wire communications system is switched off before actually executing the jump.

It is important to note that record types 08H, 0CH, 18H and 28H (used for memory reading) are not supported in 2-wire boot modes. A program extension has to be loaded somewhere else (for example in the auxiliary upper RAM located in \$FF00 to \$FFFF) to extend the supported record type set.

1.2.5. Booting from SCI

When boot1 is 1 and boot0 is 0 during the reset sequence, the extended HEX records are fetched from the RS232 serial port. Bit #8 of the address port (AD[8]) is latched upon reset into the BAUD bit of the BTREG register and is used to determine the initial baud rate of the incoming bytes: if set, $f_{osc}/4992$ is selected (3205,1 baud if a typical 16 MHz xtal is used), $f_{osc}/1536$ otherwise (10416,6 baud if a typical 16 MHz xtal is used).

After initialization, the boot program keeps reading the incoming bytes from the serial line and interpreting extended HEX records as they are completed. Once a byte has been received, the same byte is echoed back through the line.

1.3. The internal boot ROM and the auxiliary upper RAM

When booting from a serial link is selected, the internal boot ROM which stores the boot program that configures the serial link and parses the extended HEX records is enabled. It is mapped at locations \$0000 to \$01FF and, if enabled, overwrites any other

program memory that could be mapped at these locations.

In these modes the auxiliary 256-bytes RAM block mapped at addresses \$FF00 to \$FFFF is also enabled (bit #1 in RG2 set to one). Locations \$FFFD through \$FFFF of this RAM block are modified by the direct read and write commands over the internal memory. These commands are especially provided to dynamically read and write SFRs, and use self-modifying code to do direct accesses rather than indirect ones.

The normal interrupt vectors, located at addresses \$0003, \$000B and so on, are also stored in ROM and permanently point to the upper auxiliary RAM to locations \$FF03, \$FF0B and so on. No initialization is performed to these RAM locations.

Address \$FF2B is used for HEX command extension. The program branches to this address when a record type between 80H and FFH is parsed. This address is initialized with a jump to an error routine in case an extended record arrives before the extension code is downloaded. This error routine is entered whenever a colon is not found at the beginning of an HEX record, a checksum is incorrect, a record type is not recognized or any other parsing problem is detected. This routine resets the GOE bit to place the whole programmable logic in an idle mode and then keeps branching to the same address.

2. Boot program listing

We provide here a complete assemble listing of the boot program.

```
*****
*****
***** FIPSOC boot program *****
*****
***** By Julio Faura and *****
***** Ignacio Lacadena *****
*****
***** (C) SIDSA 1998 *****
*****
*****
***** FIPSOC SFRs symbol definition *****
*****

; ***** SFRs in the standard 8051 *****

SP      EQU 081H
DPL     EQU 082H
DPH     EQU 083H
PCON    EQU 087H
TCON    EQU 088H
TMOD    EQU 089H
TL0     EQU 08AH
TL1     EQU 08BH
TH0     EQU 08CH
TH1     EQU 08DH
SCON    EQU 098H
SBUF    EQU 099H
IE      EQU 0A8H
IP      EQU 0B8H
PSW     EQU 0D0H

* Definitions for bit addressing:

RI      EQU 098H    ;Receiver interrupt flag
TI      EQU 099H    ;Transmitter interrupt flag
RB8     EQU 09AH    ;9th bit (received)
TB8     EQU 09BH    ;9th bit (sent)
REN     EQU 09CH    ;Reception Enable
SM2     EQU 09DH    ;SM<2:0> == Mode specifier
SM1     EQU 09EH
SM0     EQU 09FH

; ***** Internal ports definitions *****

PORT0   EQU 80H
PORT1   EQU 90H
PORT2   EQU 0A0H
PORT3   EQU 0B0H

* Definitions for bit addressing:

RXD     EQU 0B0H    ; USART Receiver
TXD     EQU 0B1H    ; USART Transmitter

NINT0   EQU 0B2H    ; External interrupt input 0, active low
NINT1   EQU 0B3H    ; External interrupt input 1, active low
T0      EQU 0B4H    ; Timer 0 external input
T1      EQU 0B5H    ; Timer 1 external input
NWR     EQU 0B6H    ; External data memory write strobe, active low.
NRD     EQU 0B7H    ; External data memory read strobe, active low.

; ***** FIPSOC specific SFRs definitions *****
;
; The following registers are *not* present in the standard 8051.
; Therefore, some (most) macro assemblers and compilers may not
; admit them as they are not legal direct memory locations (special
; function registers) in the standard device. To avoid this problem
; they should be assembled as an external module and exported with
```

```
; GLOBAL clauses, while they should be imported with EXTERNAL clauses
; from the main module. This normally suffice to trick the assembler
; into believing that these symbols will be placed in (legal) direct
; memory locations (lower than 7FH), although the linker could object
; if it notices the final memory locations (which is not usual as the
; linker does not use to perform any consistency check).
```

```
WDOG      EQU 9AH
RG3       EQU 9BH
SCREG     EQU 9CH
I2CREG    EQU 9CH
SPIREG    EQU 9CH
BTREG     EQU 9DH
DDRP      EQU 9EH
CMBUF     EQU 9FH
```

```
VPLLL     EQU 0A4H
VPLLH     EQU 0A5H
VHW1L     EQU 0A6H
VHW1H     EQU 0A7H
```

```
VCLKL     EQU 0ACH
VCLKH     EQU 0ADH
VHW2L     EQU 0AEH
VHW2H     EQU 0AFH
```

```
VDBGL     EQU 0B4H
VDBGH     EQU 0B5H
VHW3L     EQU 0B6H
VHW3H     EQU 0B7H
```

```
VHW4L     EQU 0BCH
VHW4H     EQU 0BDH
VANAL     EQU 0BEH
VANA      EQU 0BFH
```

```
EIMR0     EQU 0C0H
EIMR1     EQU 0C1H
SGNI0     EQU 0C2H
SGNI1     EQU 0C3H
IRS        EQU 0C4H
IRSKDB     EQU 0C5H
```

```
DANA1     EQU 0D8H
DANA2     EQU 0D9H
DANA3     EQU 0DAH
DANA4     EQU 0DBH
DANA5     EQU 0DCH
DANA6     EQU 0DDH
DANA7     EQU 0DEH
DANA8     EQU 0DFH
```

```
ANAST      EQU 0E1H
DBGCNF     EQU 0E2H
DBGMSK     EQU 0E3H
DBG0L      EQU 0E4H
DBG0H      EQU 0E5H
DBG1L      EQU 0E6H
DBG1H      EQU 0E7H
```

```
DBG2L      EQU 0E8H
DBG2H      EQU 0E9H
DBG3L      EQU 0EAH
DBG3H      EQU 0EBH
DBG4L      EQU 0ECH
DBG4H      EQU 0EDH
DBG5L      EQU 0EEH
DBG5H      EQU 0EFH
```

```
CKCONF     EQU 0F1H
CKCNTL     EQU 0F2H
CKCNTH     EQU 0F3H
CKDMC2     EQU 0F4H
CKDMC1     EQU 0F5H
CKANA      EQU 0F6H
CK8051     EQU 0F7H
```



```

ROWL    EQU 0F8H
ROWH    EQU 0F9H
COLL    EQU 0FAH
COLH    EQU 0FBH
RG1     EQU 0FCH
RG2     EQU 0FDH
RGTX    EQU 0FEH
outCOMP EQU 0FFH

```

```
PAGE
```

```
TITLE "boot.asm"
```

```

;Constants for bit-addressing the accumulator:
BOOT1   EQU 7      ;These three flags store the boot mode
BOOT0   EQU 6      ;
MASTER  EQU 5      ;
BAUD     EQU 0      ;Baud rate when booting from SCI

;Constants for bit-addressing the accumulator:
CCF      EQU 7      ;Comunicación complete flag
I2CMAST  EQU 6      ;Sets to one when we are master
SDRC     EQU 5      ;1=Send, 0=receive
CK0      EQU 4      ;1=100KHz, 0=1MHz
ID       EQU 3      ;To see if we have been addressed (slave)
ERR      EQU 2      ;Sets to one when we don't get an acknowledge
CMD      EQU 1      ;Write this bit to send commands
BBUSY    EQU 0      ;Bus Busy (also used to send commands)

```

```
*****
```

```
DATA
```

```

ORG 7BH      ;This is located here so the data buffer starts on 80H

COLON        DS 1      ;Dummy buffer to store the frame start (a colon)
LENGTH       DS 1      ;Number of bytes in DATABUF
ADD_H        DS 1      ;Address high byte
ADD_L        DS 1      ;Address low byte
REC_T        DS 1      ;00 == Write to external mem
               ;08H == Read from external (data) mem
               ;0CH == Read from external program mem
               ;10H == Indirectly write to internal mem
               ;18H == Indirectly read from internal mem
               ;20H == Directly write to internal mem
               ;28H == Directly read from internal mem
               ;40H == Jump to address (LJMP)
               ;48H == Return (ends with RET instead of LJMP)
               ;Codes 80H to FFH are used for command set extension

DATABUF      DS 40H     ;Data buffer + checksum

STACK        DS 1       ;Where the stack pointer should initially point to

SFMDCOD      EQU FFFDH  ;This is the self modifying code location
               ;for SFR indirect access.
EXT_CMD      EQU FF2BH  ;Code for extending the HEX command set

COLONVAL     EQU ':'    ;Constant value for the colon character
FIRSTDEV     EQU A1H    ;The first I2C device to be probed

```

CODE

; Interrupt pseudovectors pointing to upper RAM

; Linked to INIT program (this is quite tricky!!)

ORG 00H

```
RESET    MOV R3,A           ;This clears R3 for the first TXBYTE
         SJMP INIT
```

```
VEC_IE0  LJMP FF03H        ;This should be $03
INIT     MOV SP,#STACK
         SJMP INIT1
```

```
VEC_TF0  LJMP FF0BH        ;This should be $0B
INIT1    MOV DPTR,#EXT_CMD
         SJMP INIT2
```

```
VEC_IE1  LJMP FF13H        ;This should be $13
INIT2    MOV A,#02H        ;Op-code for LJMP
         SJMP INIT3
         nop               ;Empty byte
```

```
VEC_TF1  LJMP FF1BH        ;This should be $1B
INIT3    MOV RG2,A         ;Maps the aux RAM memory for program and data
         MOVX @DPTR,A
         SJMP INIT4
```

```
VEC_RIT1 LJMP FF23H        ;This should be $23
```

; Off we go

```
INIT4    INC DPTR
         MOV A,#ERROR/100H
         MOVX @DPTR,A
         INC DPTR
         MOV A,#ERROR-(ERROR/100H)*100H
         MOVX @DPTR,A
         MOV A,SCREG        ;Strongly recommended ;)
         MOV A,BTREG        ;Select the subsystem to initialize
         JNB A.BOOT1,INIT_SCI ;If boot1==0 then we are booting from SCI
         JNB A.BOOT0,INIT_SPI ;If boot0==0 then we are booting from SPI
         JB A.MASTER,INIT_I2C ;If we are I2C master
```

; I2C configuration as slave

```
INI2C_SL MOV A,SCREG        ;Wait for BBUSY == ID == 1
         ANL A,#9
         CJNE A,#9,INI2C_SL
         ACALL TXBYTE
         MOV A,R5
         CJNE A,#89,INI2C_SL
         MOV CMBUF,#FFH     ;This is the acknowledge
         SJMP GET_FRAM
```

```

*****
; Send an STOP command and start again
*****
STOPNGO  ACALL I2CSTOP
;Uncomment the following if the next label is not INIT_I2C
;        SJMP INIT_I2C

*****
; I2C configuration as master
*****
INIT_I2C MOV A,SCREG
        JB A.BBUSY,W4BUSFRE
        ACALL START
I2CLOOP  MOV R3,#FIRSTDEV      ;We probe the first device
        ACALL TXBYTE
        JNB A.CCF,INIT_I2C
        JNB A.ERR,I2CFOUND
        INC R3                ;Next device
        INC R3
        ACALL RESTART
        SJMP I2CLOOP

I2CFOUND MOV A,R3              ;R3 has now the device that answered
        ANL A,#FEH            ;We prepare for a READ command
        MOV R3,A
        MOV CMBUF,#0          ;We send two zeroes as a 16-bit address
        ACALL WAIT4IT
        CJNE R5,#C9H,STOPNGO
        MOV CMBUF,#0
        ACALL WAIT4IT
        CJNE R5,#C9H,STOPNGO
        ACALL RESTART
        ACALL TXBYTE          ;We left the device address in R3
        CJNE R5,#C9H,STOPNGO
        SJMP GO_ON

*****
; RS232 Serial interface configuration
*****
INIT_SCI MOV SCON,#50H        ;Mode 010, REN enable, clear flags
        MOV PCON,#80H         ;Double baud rate, clear flags
        MOV TCON,#40H         ;Start the Timer #1
        MOV TMOD,#20H         ;Sets timer #1 in 8-bits reload mode
        MOV TH1,#F3H          ;243 for 3205 baud (with a 16 MHz xtal)
        JB A.BAUD,GET_FRAM     ;3205 baud if GPORT.0 is 1, 10416 baud otherwise
        MOV TH1,#FCH          ;252 for 10416 baud (with a 16 MHz xtal)
        SJMP GET_FRAM

*****
; SPI configuration
*****
INIT_SPI ANL SCREG,#F3H        ;Clears dummy flags
        JNB A.MASTER,GO_ON     ;No more initialization required if slave
        MOV PORT1,#FEH        ;Select First SPI device
        MOV DDRP,#FEH         ;Bit 0 of general port as output
        MOV R3,#3
        ACALL TXBYTE
        MOV R3,#0
        ACALL TXBYTE
        ACALL TXBYTE
;Uncomment the following if the next label is not GO_ON
;        SJMP GO_ON

GO_ON    MOV CMBUF,#FFH

*****
; Main loop for frame fetching and parsing
*****
GET_FRAM MOV R0,#COLON        ;This is the main loop (fetching records)
        ACALL GET_BYTE
        JNZ ERROR
        MOV A,R3

```

```

MOV @R0,A
CJNE @R0,#COLONVAL,ERROR ;Check if the colon was OK
INC R0
ACALL GET_BYTE
JNZ ERROR
MOV A,R3
ANL A,#3FH ;We limit the number of bytes to 3F
MOV @R0,A
ADD A,#4
MOV R2,A

GF_LOOP INC R0
ACALL GET_BYTE ;Get a byte and bring it in A
JNZ ERROR
MOV A,R3
MOV @R0,A
DJNZ R2,GF_LOOP

CHKSUM MOV R0,#LENGTH
MOV A,#4
ADD A,@R0 ;Total number of bytes = length + 4
MOV R2,A ;R2 is used as a (down) counter
MOV A,@R0

CHKLOOP INC R0
ADD A,@R0
DJNZ R2,CHKLOOP
JNZ ERROR ;Sum of al bytes + checksum = 0

MOV DPH,ADD_H ;This is going to be used in the three commands
MOV DPL,ADD_L
MOV R0,#DATABUF
MOV R1,ADD_L
MOV R2,LENGTH
MOV R4,REC_T

MOV A,R4

CJNE A,#40H,GOON1 ;Jump if REC_T == 40H
ACALL ENDCOMS
CLR A
JMP @A+DPTR
GOON1 CJNE A,#48H,GOON2 ;Return if REC_T == 48H
; ACALL ENDCOMS ;Do not stop if this is a subroutine!
RET
GOON2 JNB A.7,GOON3 ;REC_T >= 80H -> Extended command (in $FF2B)
LJMP EXT_CMD

GOON3 JNB A.3,GOON4
ACALL READMEM
JNZ ERROR
SJMP GET_FRAM

GOON4 ACALL PROGMEM
JZ GET_FRAM
; Uncomment the following if the next label is not ERROR
; SJMP ERROR

ERROR ANL BTREG,#F7H ;We reset GOE!!
SJMP ERROR

*****
; End communications
*****
ENDCOMS MOV A,BTREG
JNB A.BOOT1,ENDSCI ;If SCI, wait for TI, clear it and bail out
JNB A.BOOT0,ENDSL ;IF SPI, just switch the system off and bail out
JNB A.MASTER,ENDSL ;Otherwise, it's I2C
ACALL I2CSTOP ;If we're master, then stop the communication

ENDSL MOV PORT1,#FFH ;Otherwise, switch off the coms system
MOV DDRP,#FFH
ANL BTREG,#7FH
RET

ENDSCI JNB TI,ENDSCI

```

```
CLR TI
RET
```

```
*****
;
; Routine: PROGMEM
;
; Inputs: DPTR is ADD_H,ADD_L
;         R0 points to the first data byte in the buffer
;         R1 is ADD_L
;         R2 has the number of bytes to write
;         R4 is the record type
; Outputs: A == 0 if no error, FF otherwise
;
; Modifies: A, R0, R1, R2, DPTR
;
; Programs any kind of memory
;
*****

PROGMEM  CJNE R2,#0,PROGLOOP
         RET
PROGLOOP MOV A,@R0
         CJNE R4,#00H,PROG1 ;Program external memory
         MOVX @DPTR,A
         SJMP GOPROG
PROG1    CJNE R4,#10H,PROG2 ;Indirectly program internal memory
         MOV @R1,A
         SJMP GOPROG
PROG2    CJNE R4,#20H,RETERR;Directly program internal memory
         MOV DPTR,#SFMDCOD ;This is the self-modifying code location
         MOV A,#86H        ;Hex code for "MOV direct,@R0"
         MOVX @DPTR,A
         INC DPTR
         MOV A,R1
         MOVX @DPTR,A      ;Modify the code with the direct address
         INC DPTR
         MOV A,#22H        ;Hex code for "RET"
         MOVX @DPTR,A
         LCALL SFMDCOD

GOPROG   INC DPTR
         INC R0
         INC R1
         DJNZ R2,PROGLOOP
         CLR A             ;No error occurred
         RET

*****
;
; Routine: READMEM
;
; Inputs: DPTR is ADD_H,ADD_L
;         R0 points to the number of bytes to read
;         R1 is ADD_L
;         R2 has the number of bytes to write
;         R4 is the record type
; Outputs: A == 0 if no error, FF otherwise
;
; Modifies: A, R0, R1, R3, DPTR
;
; Reads any kind of memory
;
*****

READMEM  MOV R2,@R0        ;R0 equals #DATABUF
         CLR A
         CJNE R2,#0,READLOOP
         RET
READLOOP XCH A,R3
         CJNE R4,#08H,READ1 ;Read external memory
         MOVX A,@DPTR
         SJMP GOREAD
```

```

READ1    CJNE R4,#0CH,READ2 ;Read code
          CLR A
          MOVC A,@A+DPTR
          SJMP GOREAD
READ2    CJNE R4,#18H,READ3 ;Indirectly read internal memory
          MOV A,@R1
          SJMP GOREAD
READ3    CJNE R4,#28H,RETERR ;Directly read internal memory
          MOV DPTR,#SFMDCOD
          MOV A,#E5H          ;Hex code for "MOV A,direct"
          MOVX @DPTR,A
          INC DPTR
          MOV A,R1
          MOVX @DPTR,A
          INC DPTR
          MOV A,#22H          ;Hex code for "RET"
          MOVX @DPTR,A
          LCALL SFMDCOD        ;Now the data is in A

GOREAD    XCH A,R3
          ADD A,R3
          PUSH A                ;We save the sum
          ACALL SENDBYTE        ;We send R3
          JNZ RETERR
          POP A                 ;We restore the sum
          INC DPTR
          INC R1
          DJNZ R2,READLOOP
; Now A has the sum and we must calculate and send the checksum
          CPL A                ;Calculate checksum
          INC A                 ;'twas 2's complement
          MOV R3,A
; Uncomment the following if the next routine is not SENDBYTE
;          ACALL SENDBYTE        ;Send checksum
;          RET

*****
;
; Routine: SENDBYTE
;
; Inputs: R3 is the byte to send
; Outputs: A == 0 if no error, FF otherwise
;
; Modifies: A, R5
;
; Sends a byte no matter where thru. It waits for a previous
; transmission to be completed:
; - If SCI is used, then it waits for a serial byte to come
;   prior to send R3.
; - If SPI is used, then it waits for SPIF to be up. If we
;   are slave, they should provoke a (dummy) transmission
;   before we send R3; if we are master, we should provoke
;   a dummy transmission before calling SENDBYTE. This is
;   normally used for sending a group of bytes, so each
;   transmission is used to trigger the following. Obviously
;   the first transmission has to be provoked manually.
;
*****

SENDBYTE MOV A,BTREG
          JNB A.BOOT1,SB_SCI
          JNB A.BOOT0,SB_SPI
          MOV A,#FFH          ;I2C reads not supported
          RET

SB_SPI    ACALL WAIT4IT        ;We wait for the previous transmission to end
          CJNE A,#80H,RETERR    ;Maybe we shouldn't check this out
          MOV CMBUF,R3          ;We send R3
          CLR A                 ;No error occurred
          RET

SB_SCI    JNB RI,SB_SCI        ;We wait for the previous transmission to end
          CLR RI                ;We assume nothing is left
          CLR TI
          MOV SBUF,R3          ;We send R3

```



```

CLR A                ;No error occurred
RET

*****
; Return with A == FF (error)
*****
RETERR    MOV A,#FFH
          RET

*****
;
; Routine: GET_BYTE
;
; Inputs: None
; Outputs: A stores bits CCF and ERR
;          R3 stores the received byte
;          R5 stores SCREG but bits SDRC and CK0
;
; Modifies: A, R3, R5 (WAIT4IT is called afterwards)
;
; Gets a byte no matter where from and echoes it afterwards
;
*****

GET_BYTE  MOV A,BTREG          ;Get a byte no matter where from, result in R3
          JNB A.BOOT1,GB_SCI

GB_SCOM   ACALL WAIT4IT        ;Get a byte from I2C or SPI, master or slave
          CJNE A,#80H,RETERR
          MOV R3,CMBUF
          MOV CMBUF,R3
          CLR A                ;No error
          RET

GB_SCI    JNB RI,GB_SCI        ;Get a byte from the serial port
          CLR RI
          MOV R3,SBUF
          CLR TI
          MOV SBUF,R3          ;Echo the received byte
          CLR A                ;No error
          RET

*****
;
; Routine: TXBYTE
;
; Inputs: R3 is the byte to send
; Outputs: A stores bits CCF and ERR
;          R5 stores SCREG but bits SDRC and CK0
;
; Modifies: A, R5 (WAIT4IT is called afterwards)
;
; Transmits a byte through the I2C or SPI line and waits until
; transmission is finished
;
*****

TXBYTE    MOV CMBUF,R3         ;Send a serial byte (stored in R3)
; Uncomment the following if the next routine is not WAIT4IT
;          ACALL WAIT4IT
;          RET

*****
;

```

```

; Routine: WAIT4IT
;
; Inputs: None
; Outputs: A stores bits CCF and ERR
;         R5 stores SCREG but bits SDRC and CK0
;
; Modifies: A, R5
;
; Waits for a transmission to be completed (CCF or ERR)
;
*****

WAIT4IT  MOV A,SCREG          ;Wait for a serial byte to be transmitted
        ANL A,#CFH           ;Bits 5 and 4 are not interesting
        MOV R5,A             ;Save SCREG in R5
        ANL A,#84H           ;Wait for CCF or ERR
        JZ WAIT4IT
        RET

*****

;
; Routine: I2CSTOP
;
; Inputs: None
; Outputs: None
;
; Modifies: A and R2 (calls SENDCMD)
;
; Sends a STOP command to the I2C interface (using SENDCMD)
;
*****

I2CSTOP  MOV R2,#2
        ACALL SENDCMD
        RET

*****

;
; Routine: START, RESTART
;
; Inputs: None
; Outputs: None
;
; Modifies: A, R2 (calls SENDCMD)
;
; Sends a START command to the I2C interface (using SENDCMD)
;
*****

START
RESTART  MOV R2,#3
; Uncomment the following if the next routine is not SENDCMD
;         ACALL SENDCMD
;         RET

*****

;
; Routine: SENDCMD
;
; Inputs: R2 is used to specify the command
;         use XXXX_XX11b for START and RESTART
;         use XXXX_XX10b for STOP
; Outputs: None
;
; Modifies: A, R2
;
; Used for sending a STOP, START or RESTART command to the
; I2C interface
;
*****

SENDCMD  MOV A,SCREG

```

```
ANL A,#FCH
ORL A,R2
MOV SCREG,A
CMDLOOP MOV A,SCREG
JB A.CMD,CMDLOOP
```

```
RET
```

```
end
```