In-Application Programming for the ST92F120

by Microcontroller Division Application

INTRODUCTION

In-Application Programming (IAP) is used to update the contents of the Flash memory in the field without the use of any special hardware tools. To update firmware, the user must run the bootloader that will download the new firmware to the Flash memory.

Once programmed, the bootloader remains permanently in the Flash memory.

The firmware, or user code, is the code programmed in the Flash memory by the bootloader.

This application note gives an example of how to implement a bootloader program on a ST92F120 microcontroller and provides guidelines on how to customize this bootloader for your application needs.

1 EXAMPLE CODE

The bootloader provided with this application note is used to update Flash and EEPROM memory contents by receiving a standard Intel® hex file via a RS-232 serial link.

In order to update the code, a terminal program able to send text files is required. The format used by the serial link is 9600 bauds, 8 data bits, 1 stop bit, no parity and no flow control.

It takes about 5 minutes to update the entire user available memory (120 Kbytes of Flash and 1 Kbyte of EEPROM).

1.1 GENERAL ORGANIZATION

1.1.1 Main Modules

Figure 1 shows the general organization of the provided bootloader:

– **communication.c** module handles RS-232 data transfers,
– **iap.c** is the main module. It interfaces with the communication.c module to obtain data and with the pgmmem.c module to program the memory,
– **pgmmem.c** module handles Flash and EEPROM operations,
– **msg.c** module is a basic message, warning and error handler.

Figure 1. General Organization
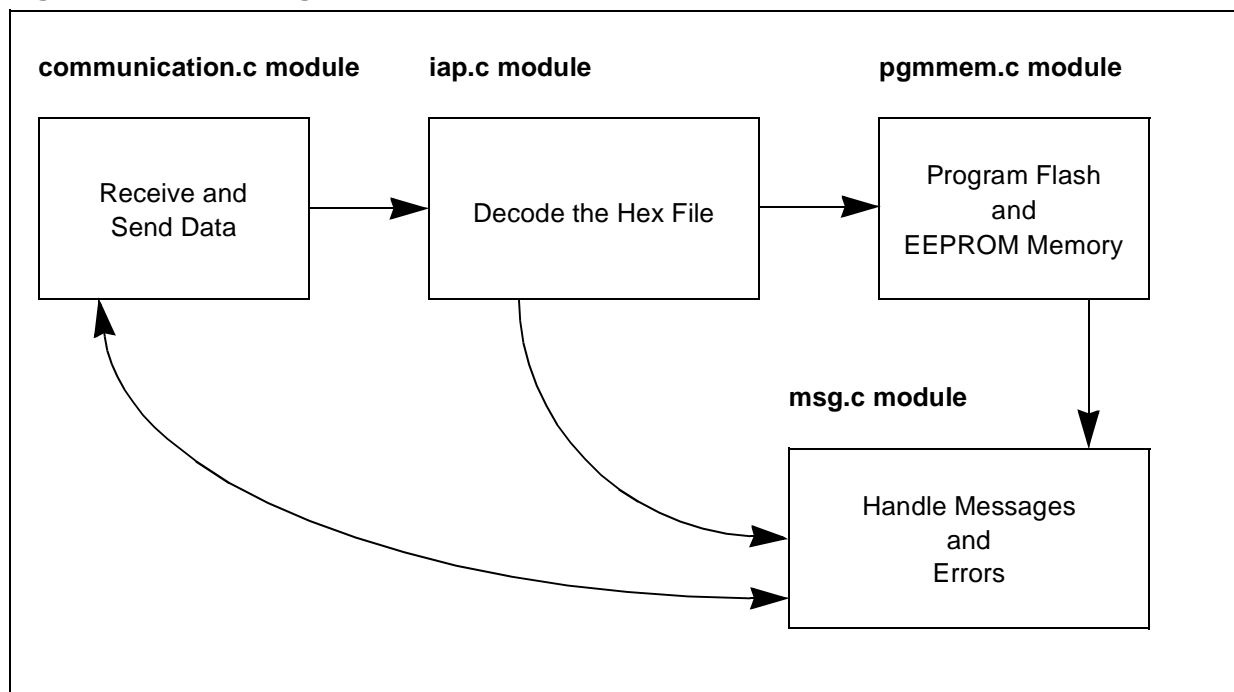
1.1.2 Workspace

A diagram describing the project workspace is shown in Figure 2.

Each directory is associated with a given file type:

   – **pjt** are files related to the project,
   – **c** are main module c source files,
   – **spp** are SPP startup files, low level memory programming functions and interrupt vectors,
   – **include** are public function declaration files, memory mapping (device.h), I/O port macro instructions (ioinit.h) and general macro instructions (define.h).

Figure 2. Project Workspace
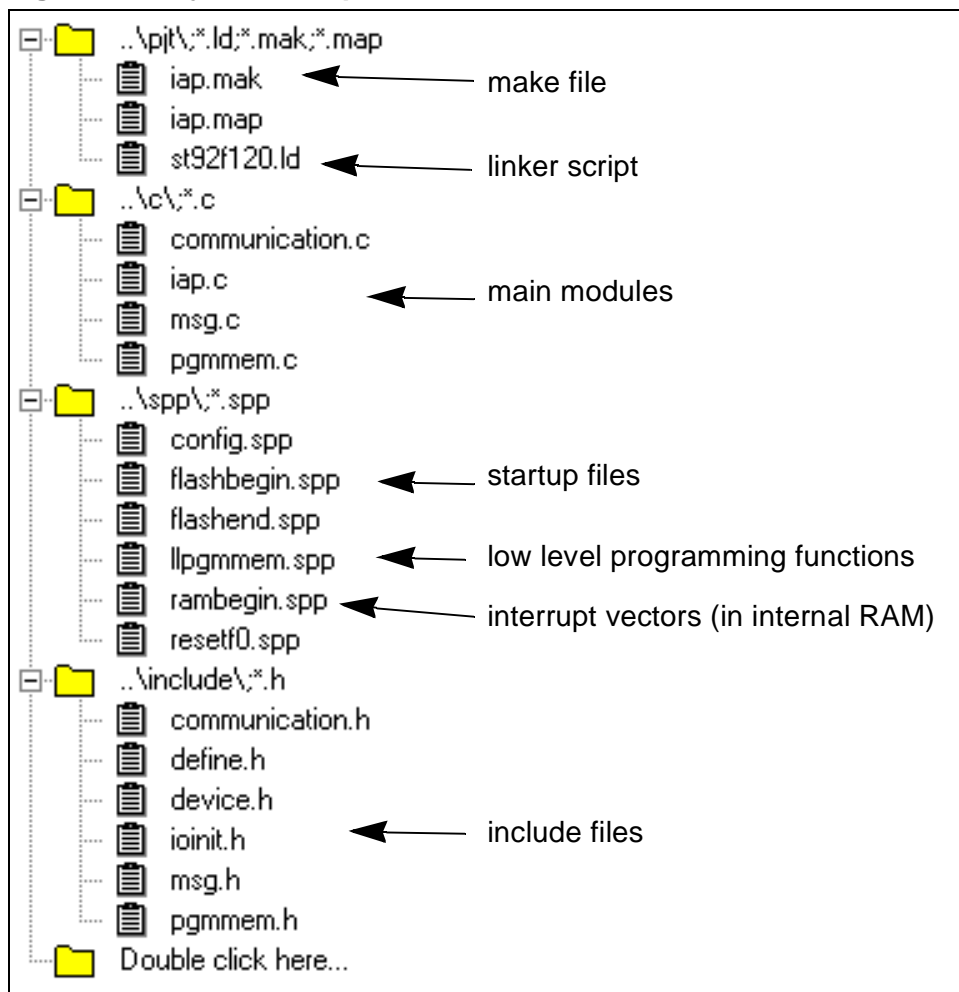
1.2 DETAILED IMPLEMENTATION

1.2.1 ST92F120x1 constraints

The bootloader provided is targeted for the ST92F120x1 (ST92F120 MCU with 128K of Flash memory) microcontroller. This section describes the specific features of the ST92F120x1 device.

1.2.1.1 Resources Reserved by the Bootloader

After a reset, the bootloader program must always be executed first in order to ensure that the user code is valid before it is given control. To ensure that the bootloader is executed first, it must be located in the same Flash memory sector as the Reset vector. On ST92F120x1 MCUs, this is sector 3 starting from address 0x1e000, which is then no longer available for user code.

The user code will be considered as being valid if the previous update occurs without any errors being detected. The bootloader must check a flag in EEPROM in order to know if the firmware is valid or not: this flag is set to "P" at the start of each update and to "E" at the end if no errors have been detected. The firmware is then valid when the flag equals "E" at start-up. The byte use for this flag is located at address 0x2203ff in EEPROM and is no longer available for the user application. This flag is referred to as the successful memory programming flag.

1.2.1.2 Memory Protection

The ST92F120 supports two kinds of memory protection, the first one for both read and write access and the second one for write access only. Typically, read and write access rights are protected to prevent any unwanted memory dumps. Write-only access rights are protected to prevent any spurious write access to the memory.

Before updating the memory, all active protections have to be temporarily disabled.

1.2.1.3 Memory Programming

The MCU memory is programmed by downloading the new firmware into the program memory. This firmware is written into the ST92F120 Flash memory by the bootloader program. The bootloader performs all Flash memory erase and write operations from the internal RAM because instructions can no longer be fetched from Flash memory when Flash erase or write operations are in progress.

The EEPROM can be programmed from Flash memory, but its programming cycle is much slower than the Flash programming cycle: a maximum of 100 ms for the EEPROM page update cycle against a maximum of 1.2 ms for Flash byte programming cycle.

1.2.1.4 Interrupts

The interrupt vectors are the first 256 bytes of the ISR segment. As the bootloader location in sector 3 (addresses 0x1e000 to 0x1ffff) does not include the first 256 bytes of segment 1, interrupt vectors are located at the beginning of segment 0x20 in the internal RAM.

Furthermore, it is possible to enable interrupts during Flash write or erase operations only if the interrupt vectors and routines are located outside the Flash, as this memory cannot be read while a write or erase operation is in progress.

1.2.2 communication.c Module

This module handles SCI data transfers. A double buffer mechanism is used for reception to avoid using flow control and a simple pooling mechanism is used for transmission.

1.2.2.1 Public Functions

void Init_Communication(void)

> This function initializes the Serial Communications Interface (SCI), the DMA address and counter pointers and the timeout counter. The SCI character format is 9600 bauds, 8 data bits, 1 stop bit and no parity.

unsigned char Get_Char(void)

> This function returns a data byte from the buffer if the end of file is not yet reached, otherwise it returns ":".

*void Put_Msg(const char * msg)*

> This function sends each byte of the *msg* string to the *Put_Char* function.

void Communication_Periodical_Function(void)

> This function is called every milli-second by the iap.c module. If a DMA End of Block interrupt has not been detected since the milli-second *TIMEOUT* period (4 times the time needed to receive an entire buffer), the current buffer status is set to *HALF_FULL* meaning that the end of file is reached.

1.2.2.2 Private Functions

void It_Dma_Eob(void)

> This is the DMA end of block interrupt. At the end of each DMA transfer, it swaps one buffer to the other and refreshes the timeout counter.

void It_Receiver_Error(void)

> This interrupt generates an error message whenever any reception errors are detected.

void Put_Char(unsigned char character)

> This function sends the byte character through the SCI.

1.2.2.3 Operation

After the *Init_communication* function has initialized the SCI and the double buffer mechanism, data is ready to be received. Flow control is not required since the 9600-baud transfer rate is slow enough for the Flash memory to be programmed at the same time that the data is received. (In order to program one byte, 2 ASCII bytes must be received. This takes approximately 2 ms at 9600 bauds, while the Flash programming cycle does not exceed 1.2 ms.)
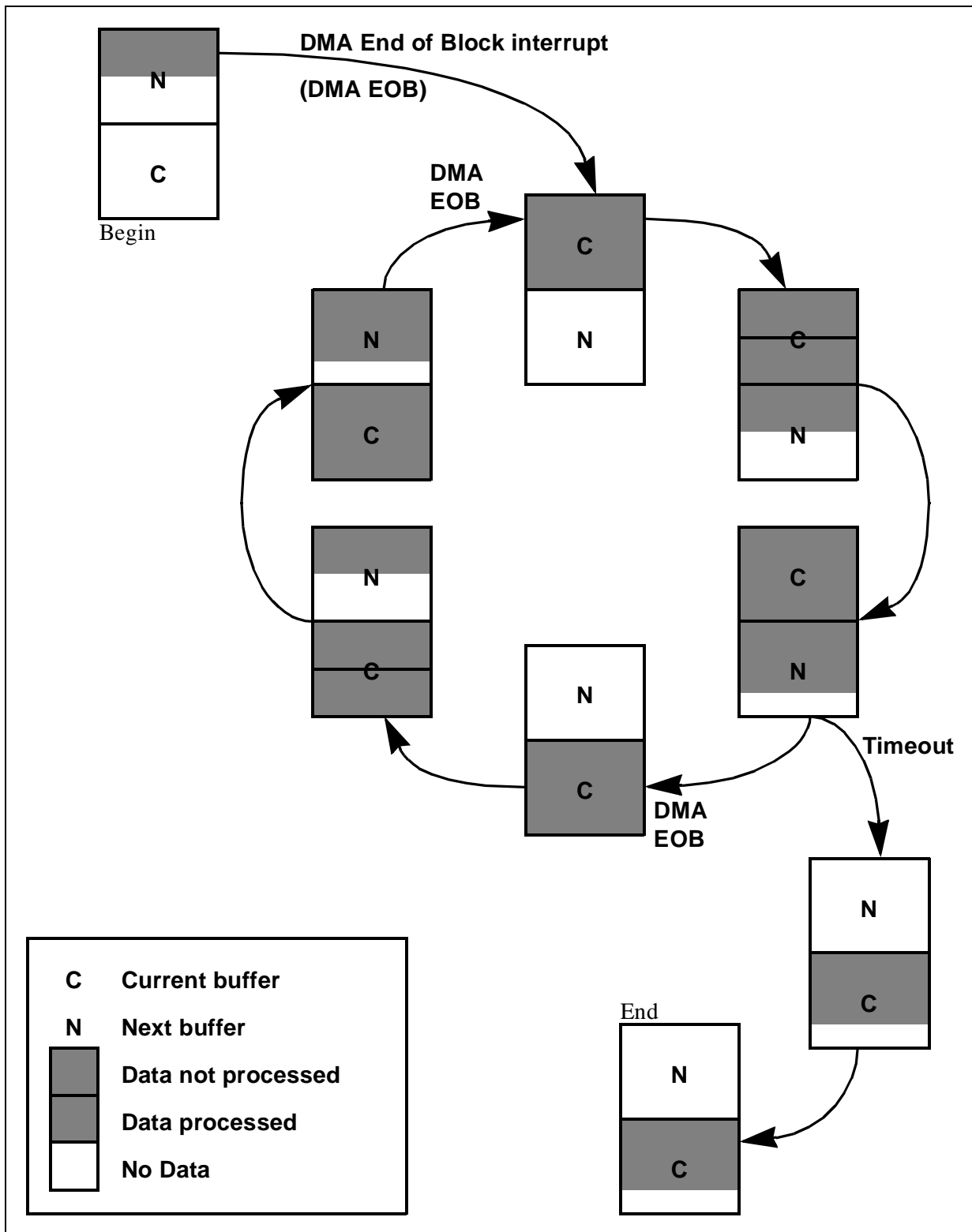
The double buffer mechanism is illustrated in Figure 3.

The *Get_Char* function obtains data from the current buffer while the next buffer is being filled. As the Flash programming cycle is faster than the 2-byte reception rate, current buffer data will be entirely processed before the next buffer is full. When the next buffer is full, it becomes the current buffer and then the data source for the *Get_Char* function.

When the end of file is reached, the next buffer is not necessarily full. This is why a timeout counter is implemented; e.g. if a DMA End of Block interrupt has not been detected since millisecond *TIMEOUT* period (4 times the time needed to receive an entire buffer), the next buffer becomes the current buffer and its status is set to *HALF_FULL*.

The *Put_Msg* function is used to output a string through the SCI. Each byte is sent using the *Put_Char* private function.

Figure 3. Double Buffer Mechanism

DMA End of Block interrupt

(DMA EOB)

DMA EOB

Begin

Timeout

DMA EOB

C	**Current buffer**
N	**Next buffer**
	Data not processed
	Data processed
	No Data

End

1.2.3 iap.c Module

This module is used to obtain the hex file via the communication.c module and then decodes it and initiates the memory programming procedure via the pgmmem.c module.

Iap.c is the main module and only contains private functions.

1.2.3.1 Private Functions

void Isp_Main(void)

> This is the main function of the iap.c module. After initialization, this function enters a loop which decodes the hex file and initiates memory programming by calling the *Write_Memory* function of the pgmmem.c module.

void Init_Periodical_Interrupt(void)

> This function initializes the generation of an interrupt by the standard timer at the *IT_FREQUENCY* rate.

void It_Periodical(void)

> This interrupt is called at an *IT_FREQUENCY* rate in order to manage internal and external watchdog refresh operations and the timeout counter for the communication.c module.

void Refresh_Wdt(void)

> This function is called by the *It_Periodical* function and is used to refresh the internal watchdog. If an external watchdog is used, its refresh code should be located in this function.

> **Note:** The external watchdog refresh code must be located in RAM as it can be called when a Flash write or erase operation is in progress.

unsigned char Get_Hex_Byte(void)

> This function returns a hexadecimal byte from the hex file.

unsigned int Get_Hex_Word(void)

> This function returns a hexadecimal 16-bit word from the hex file.

unsigned char Ascii_To_Hex(unsigned char ascii)

> This function converts ASCII bytes from the hex file into hexadecimal nibbles.

unsigned long Get_Linear_Address(unsigned int ulba, unsigned int usba, unsigned int hex_offset)

> This function returns the linear address of a data byte. For more information, please refer to the hexadecimal object file format from Intel®.

void Verify_Checksum(void)

> This function is called at the end of each hex file line processed in order to verify the checksum. An error is generated if the checksum is not valid.

void Init_Pll(void)

> This function turns on the PLL that enables a 12 MHz internal clock using a 4 MHz external crystal or oscillator.

1.2.3.2 Operation

On start-up, the *Isp_Main* function refreshes the watchdog, turns on the PLL and sets up the standard timer to generate a periodical interrupt. The *Isp_Main* function calls the *Unprotect_Memory* and *Init_Eeprom* functions of the pgmmem.c module so that the testflash and Flash memory can be accessed and to initialize the EEPROM RAM buffer. The *Isp_Main* functions calls the *Init_Communication* function of the communication.c module in order to initialize the SCI and DMA transfers.

Before any memory erase or write operation takes place, a successful memory programming flag is set to "P" in EEPROM, meaning that the device is in a programming state.

Then, user available flash sectors are erased and the program enters the main loop.

The main loop decodes the hex file and initiates memory programming by calling the *Write_Memory* function of the iap.c module until the End of File record is encountered in the hex file.

At the end of the main loop, the *Update_Eeprom* function of the pgmmem.c module is called to update the contents of the EEPROM.

Finally, the successful programming flag is set to "E", meaning that the Flash memory has been updated and no errors have been detected. This flag is checked at start-up and the control is given to the user program only if this flag is set to "E", meaning that the last memory update was successful.

1.2.4 pgmmem.c Module

This module handles Flash erase and write operations and EEPROM write operations.

1.2.4.1 Public Functions

void Write_Memory(unsigned long linear_address, unsigned char data)

> This is the main function of the pgmmem.c module. According to the *linear_address* linear address, the *data* byte is written either to the Flash or EEPROM buffer or the write operation is discarded if the *linear_address* address does not belong to one of the Flash or EEPROM address spaces.

void Unprotect_Memory (void)

> This function temporarily disables the access protection of the user available Flash sectors.

> **Note:** This function does not temporarily disable EEPROM access protection. It is up to the user to disable this protection if needed.

void Erase_Flash_Sector(unsigned char sector)

> This function erases the *sector* Flash sector if this sector is user available (sectors 0, 1 and 2), otherwise the Flash erase operation is discarded. The control is given back to the caller at the end of the erase operation.

void Write_Flash(unsigned long linear_address, unsigned char data)

> This function writes the *data* byte at the *linear_address* address in Flash memory. The control is given back to the caller at the end of the write operation.

boolean Is_Flash_Busy(void)

> This function returns TRUE if a Flash erase or write operation is in progress, otherwise it returns FALSE.

void Init_Eeprom(void)

> This function copies the EEPROM content to an internal RAM buffer. This function must be called before any call to the *Write_Memory* function.

void Update_Eeprom(void)

> This function updates the EEPROM content from the RAM buffer. This function must be called after all calls to the *Write_Memory* function have been made.

void Write_Eeprom(unsigned long linear_address, unsigned char data)

> This function writes the *data* byte in EEPROM memory at the *linear_address* address. The control is given back to the caller at the end of the write operation.

boolean Is_Eeprom_Busy(void)

> This function returns TRUE if an EEPROM write operation is in progress, otherwise it returns FALSE.

1.2.4.2 Private Functions

void Write_Eeprom_Buffer(unsigned long linear_address, unsigned char data)

> This function is called by the *Write_Memory* function when accessing the EEPROM address space in order to update the RAM buffer.

void Unprotect_Memory_From_Flash(void)

> This function is called by the *Unprotect_Memory* function to temporarily disable the Flash access protection. This function is located in the Flash memory.

void Unprotect_Memory_From_Ram(void)

> This function is called by the *Unprotect_Memory* function to temporarily disable the Flash and EEPROM write protections. This function is located in internal RAM.

1.2.4.3 Operation

Write_Memory is the main function used to write a data byte to the Flash or EEPROM memory. The *Write_Memory* function calls the appropriate function to write either to the Flash or EEPROM RAM buffer according to the memory space pointed by the linear address. The write operation will be discarded if the linear address points to testflash, OTP or bootloader memory space.

The SCI baud rate is calculated in order to enable the Flash memory to be programmed at the same time that the data is received. The EEPROM programming cycle is much longer than the Flash programming cycle and therefore the EEPROM cannot be programmed at the same time it receives data. This is why an EEPROM RAM buffer is used, as illustrated in Figure 4.

Before the *Write_Memory* function can be called, the *Init_Eeprom* function must be called in order to copy the EEPROM contents into the RAM buffer. Any subsequent *Write_Memory* calls targeted to EEPROM will update the EEPROM image in RAM. Once the hex file is received, the *Update_Eeprom* function updates the EEPROM contents from the RAM buffer.

Figure 4. EEPROM RAM Buffer
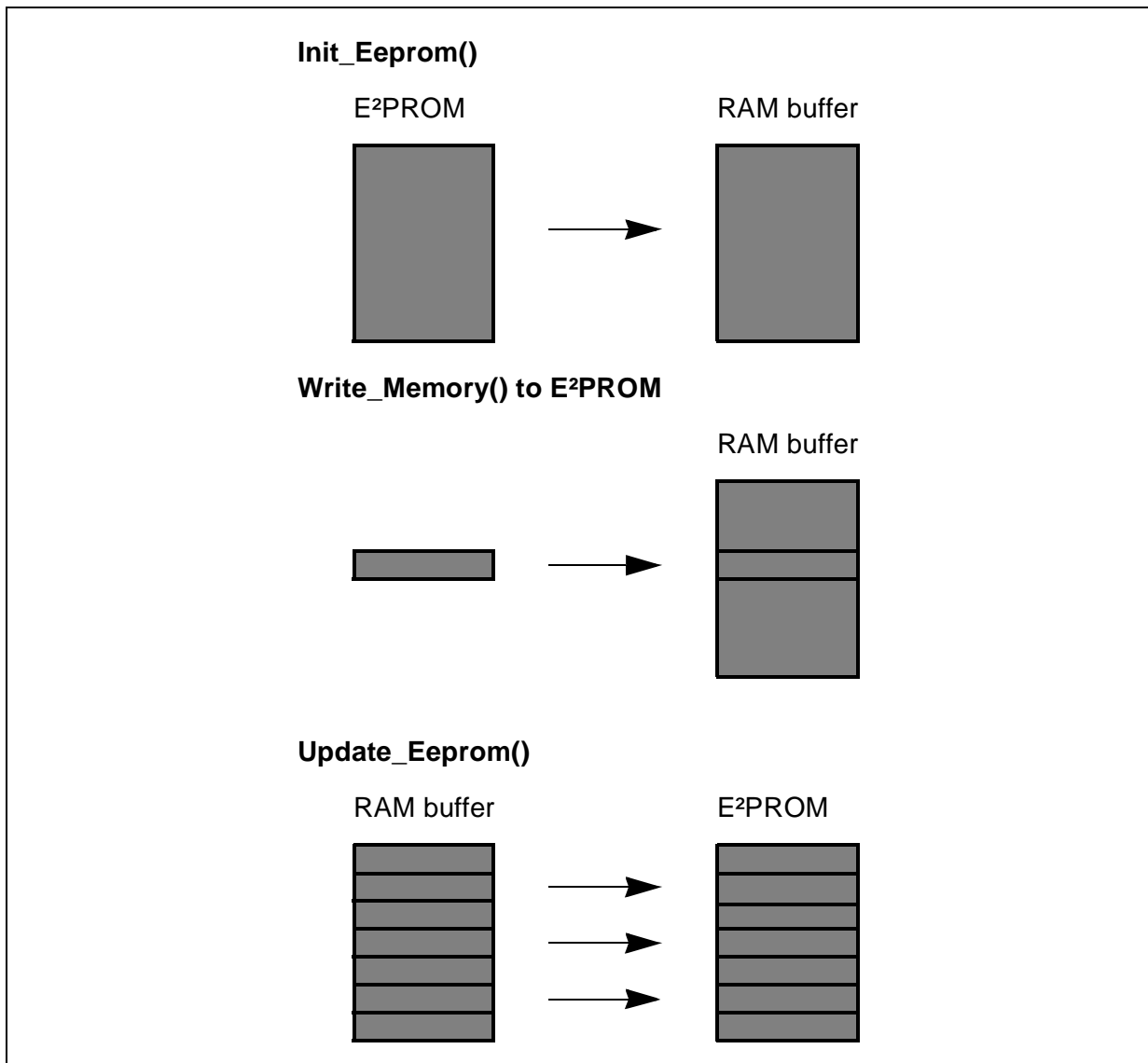
1.2.5 llpgmmem.spp Module

This module contains the low level memory writing and reading functions used by the pg-mmem.c module.

void LLWrite_Memory(unsigned long linear_address, unsigned char data)

> This function performs the "LD (*linear_address*), *data*" function that is used during Flash or EEPROM write operations.

unsigned char LLRead_Memory(unsigned long linear_address)

> This function returns the byte located at the *linear_address* address.

1.2.6 msg.c Module

This module is a very basic message, warning and error handler.

*void Msg_Manager(const char *msg, T_msg msg_type)*

> This function calls the *Put_Msg* function of the communication.c module in order to send the message through the SCI and set the microcontroller to Halt mode if the message type equals ERROR.

1.2.7 flashbegin.spp File

This is the start-up file, derived from the standard crtbegin.spp file of the v6 software toolchain. The flashbegin.spp file is executed first on start-up and gives control to the user program only if the user reset vector is programmed and the successful programming flag is set to "E", otherwise the bootloader is executed.

1.2.8 rambegin.spp File

This file contains the interrupt vectors. On bootloader start-up, the interrupt vectors are copied to the internal RAM by the flashbegin.spp module.

1.3 HOW TO USE THE BOOTLOADER

1.3.1 User Code Requirements

The user code must not use any resources reserved by the bootloader, i.e. Flash sector 3 and the successful programming flag located at address 0x2203ff in EEPROM.

The user reset vector must be located at the beginning of Flash sector 0 (address 0x000000).

1.3.2 Running the Bootloader

The bootloader is executed after a reset if no user code is programmed in memory or if an error was detected during the last update. It is considered that there is no user code programmed in memory when the user reset vector is left unprogrammed (0xffff) and that an error was detected during the last update if the successful programming flag does not equal "E".

The bootloader can also be called from the user code to perform a firmware upgrade. To run the bootloader, the user code must perform a long jump at address 0x01e004 (**asm ("jps 0x1e004");** in c source code).

Note: Before giving the control to the bootloader, the user code must disable all previously enabled interrupts by resetting all the interrupt mask bits that were previously set and by clearing the IEN bit of the CICR register.

1.3.3 Sending the Hex File

A terminal program is required to send the new firmware through the serial port. The character format used is 9600 bauds, 8 data bits, 1 stop bit and no parity. The new firmware that is to be programmed into the memory must be in standard Intel® hex format.

The device is ready to receive the new firmware when the "Ready to receive the hex file" message is displayed by the terminal program as illustrated in Figure 5. To send the hex file, set the parameters in the terminal program to send a text or ASCII file.

Figure 5. Terminal Program Screenshot

```
SCI : initialization OK
***** IAP version 1.00 *****
Erasing flash sector 0
Erasing flash sector 1
Erasing flash sector 2
Ready to receive the HEX file
```

If no errors have been detected during programming, the "End of programming" message is displayed and the device is reset, otherwise an error message is displayed and the device enters Halt mode and waits for an external reset.

1.3.4 Debugging

As the bootloader and the user code are two different applications, they cannot be both fully debugged at the same time. The debug information must be selected for either the bootloader or the user code.

Note: The *#define EMULATOR* line in the define.h module must be uncommented when using the emulator in order to remove the call used to disable the test Flash access protection function and to redirect the emulator reset vector (address 0x000000) to the device reset vector (address 0x01e000).

1.3.4.1 Using the Bootloader Debug Information

To use bootloader debug information, open the bootloader workspace and switch to Debug mode.

To emulate a firmware download to a blank memory location, you simply have to run the code.

To emulate a firmware update to an already programmed part, import the hex file of the older firmware to memory before starting the debug procedure. In this case, the user code will be executed before the bootloader on start-up because the emulator reset vector is located at address 0x000000.

Note: To import a hex file to memory, using the mouse, right-click in the memory window and open the option File>Restore Layout... and select the hex file to be imported.

1.3.4.2 Using User Code Debug Information

To use user code debug information, open the user code workspace and switch to Debug mode.

Before starting to debug the user code, import the bootloader hex file to memory. In this case, the user code will be executed before the bootloader on start-up because the emulator reset vector is located at address 0x000000.

Note: To import a hex file to memory, using the mouse, right-click in the memory window and open the option File>Restore Layout... and select the hex file to be imported.

2 BOOTLOADER CUSTOMIZATION

The purpose of this section is to help users customize the provided bootloader according to their application needs. In most cases, only the communication.c module will be modified by the user in order to receive data from a peripheral other than the SCI.

2.1 COMMUNICATION.C CUSTOMIZATION

When using a peripheral other than the SCI, the communication.c module must be modified. The first requirement is to ensure that the following public functions are included in the new communication.c module:

void Init_Communication(void)

> This function is called by the iap.c module before any data is transferred in order to initialize the communication peripheral as well as the variables used by the communication.c module.

unsigned char Get_Char(void)

> This function is called by the iap.c module to obtain a data byte from the communication peripheral. This function gives control back to the caller only when a byte is available. If no bytes are available, this function must return ":".

*void Put_Msg(const char * msg)*

> This function is called by the bootloader to display a message. It is up to the user to decide how to handle the messages: send them through the peripheral, switch ON/OFF control LEDs,...

void Communication_Periodical_Function(void)

> This function is called every milli-second from the iap.c module. It may be used to detect the End of File by implementing a timeout counter when using the DMA transfer.

> **Note**: This function must be located in RAM.

The second requirement is to locate all the functions either in RAM or in the bootloader Flash sector: all other Flash sectors are erased before starting to download the new firmware. Consequently, communication functions cannot be updated through the bootloader.

2.2 CUSTOMIZATION OF OTHER MODULES

The communication.c module is the module most likely to be modified, but other modules can also be modified. This may be required, for example, in order to handle a different object file format.

The following rules must be followed when modifying modules:

– Interrupt routines and functions called by interrupt routines must be located in RAM,
– All other functions used by the bootloader must be located in the bootloader Flash sector,
– Public functions must remain available to other modules (unless other modules are modified and they are no longer required).

"THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNEXION WITH THEIR PRODUCTS."