AN2659
Application note

STM8S in-application programming (IAP) using a customized bootloader

Introduction

This application note is intended for STM8 firmware and system designers who need to implement an In-Application Programming (IAP) feature in the product they are developing with the STM8S microcontroller.

The STM8S is an 8-bit microcontroller family with Flash memory for user program code or firmware.

IAP makes it possible to update the firmware ‘in situ’, after the microcontroller has been embedded in the final product. The advantage is that the microcontroller board can stay inside its product enclosure, no mechanical intervention is needed to do the update.

IAP is extremely useful for distributing new firmware versions. It makes it easy to add new product features and correct problems throughout the product life cycle.

The bootloader firmware source code provided with this application note shows an example of how to implement IAP for the STM8 microcontroller. You can use this code as a reference when integrating IAP in your STM8 application.

Contents

1 **Theory of operation 3**
1.1 Block versus word programming 3
1.2 RAM versus Flash programming code location 3
1.2.1 Programming the data EEPROM area 4
1.3 Flash memory protection, bootloader 4
1.4 Library support for Flash programming 5

2 **Library support for IAP 6**
2.1 Flash programming function list 6

3 **Compiler support (Cosmic) for RAM code execution 8**
3.1 Description how to compile/link code into RAM address locations 8

4 **STM8 devices with boot ROM - built-in IAP implementation 10**
4.1 Implementation details 10
4.2 Bootloader protocol 10

5 **Example of user bootloader application 11**
5.1 Bootloader firmware description 11
5.2 Used library functions 14
5.3 Block programming - RAM code copy 14
5.4 Interrupt vector table redirection 14
5.5 Bootloader overwrite protection 14
5.6 Used bootloader interface 14

6 **Conclusion 16**
6.1 Features in final bootloader application 16

7 **Revision history 17**

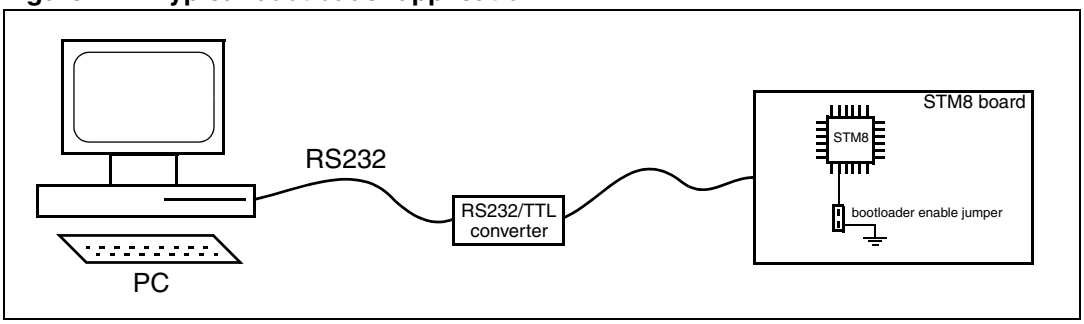
1 Theory of operation

In practice, IAP (In Application Programming) requires a bootloader implemented in the STM8 firmware that can communicate with an external master (such as a PC) via a suitable communications interface. The new code can be downloaded into the microcontroller through this interface. The microcontroller then programs this code into its Flash memory.

The STM8 microcontroller family has IAP support in hardware. The software library available for the STM8 family contains support for IAP programming. This simplifies implementation of IAP in the end-user application.

IAP programming can also be used to update the content of the internal Data EEPROM memory, and the internal RAM memory.

Figure 1. Typical bootloader application



1.1 Block versus word programming

STM8 family microcontrollers contain Flash type program memory where firmware can write. There are two methods for writing (or erasing) Flash program memory:

- Byte/word programming (1 or 4 bytes)
  - Advantages: offers small area programming, programming is done from Flash
  - Disadvantages: program stops during programming, programming speed is slow
- Block programming (128 bytes - or one Flash block for a given STM8 type)
  - Advantages: offers large area programming with high speed (large blocks)
  - Disadvantages: programming routine must run from RAM (need to copy programming routine into RAM)

1.2 RAM versus Flash programming code location

Depending on the selected programming method (see *1.1: Block versus word programming*) the programming code must run from RAM or from Flash memory.

If the programming code runs from Flash, then we can use only word/byte programming to program the Flash memory. During Flash memory programming, the code cannot access the Flash memory (Flash is in programming mode). Therefore program execution from Flash is stopped during programming (for several milliseconds) and then continues. This mode is useful in cases where only a small part (a few bytes) of Flash memory need to be updated or when it does not matter that programming is (very) slow.

To program a large Flash memory area with optimum speed then we should use block programming mode. Block programming mode can be performed only by code located in RAM. First we must copy the programming code into RAM and then run (jump to) this code. The RAM code then uses block mode to program the Flash. In this mode, programming one block takes the same time as programming one byte/word in byte/word mode. As a result programming speed is very fast and code execution is not stopped (because it is running from RAM). The only disadvantage of this method is the RAM code management:

- Copying the executable code to RAM
- Storing the RAM code (usually stored in Flash - but can be downloaded from outside)
- Allocating RAM space for code
- Compiling the code to be able to run from RAM

1.2.1 Programming the data EEPROM area

In case of data EEPROM programming, the programming code does not have to be executed from RAM. It can be located in Flash program memory, even if block programming is used. This Read-While-Write (RWW) feature speeds up microcontroller performance during IAP. Only the data loading phase must execute from RAM - this is the part of code which loads data into the EEPROM buffer. However during the physical programming phase (which takes several milliseconds) the code runs from Flash while the data EEPROM memory is programmed in the background. Completion of data EEPROM programming is indicated by a flag. An interrupt can be generated when the flag is set.

1.3 Flash memory protection, bootloader

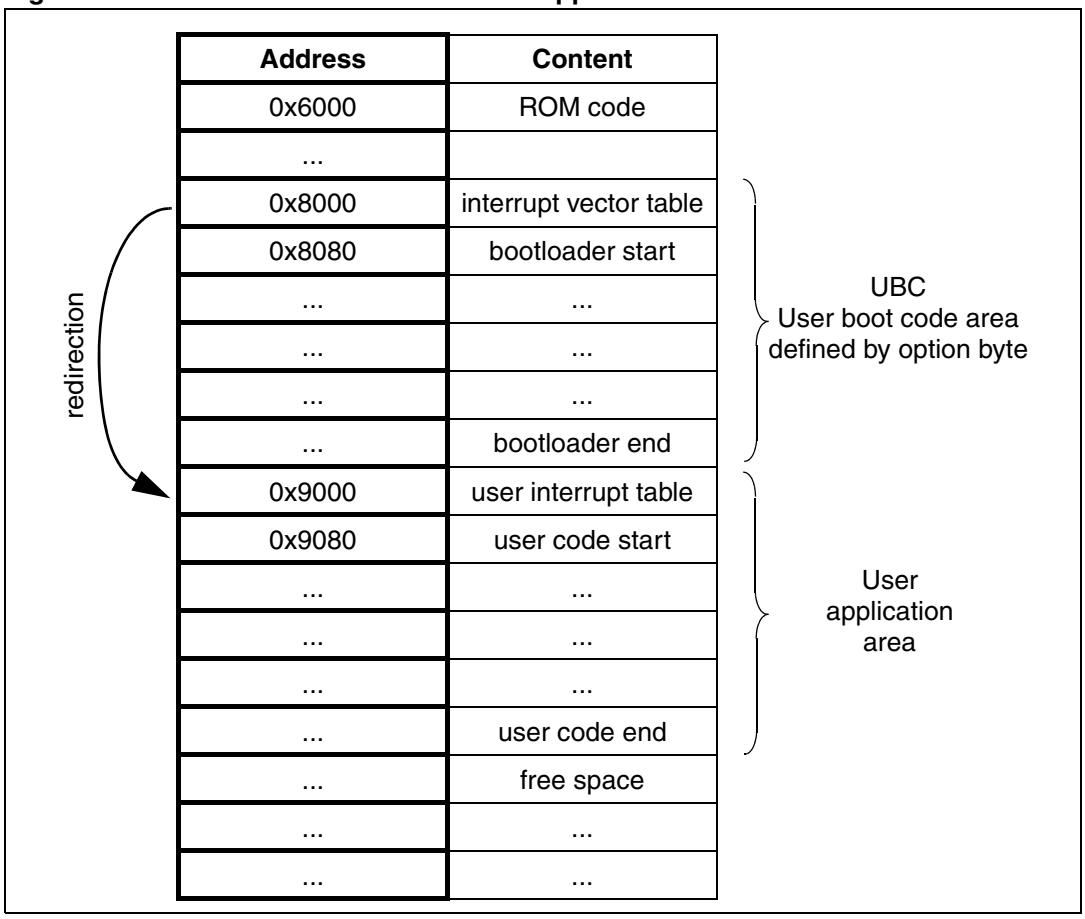
To avoid accidental overwriting of Flash code memory (for example in the case of a firmware crash) several levels of write protection are implemented in the STM8 microcontroller family.

After an STM8 reset, write access to Flash memory is disabled. To enable it, firmware must write 2 unlock keys in a dedicated register. If the unlock keys are correct (0x56, 0xAE) then write access to Flash memory is enabled and it is possible to program Flash memory using either byte/word or block programming mode. If the unlock keys are incorrect, then write access to Flash memory is disabled until the next device reset. After writing to Flash memory, it is recommended to enable write protection again by clearing a specific bit in the Flash control register (to avoid accidental write).

A similar protection mechanism exists for Data EEPROM memory, with a specific unlock register and unlock keys (0xAE, 0x56).

Additional write protection exists for programming code itself, to avoid overwriting critical code. It is designed to protect the bootloader code from being overwritten during IAP. The STM8 family has a user boot code (UBC) area which is permanently write protected. This UBC area starts from the Flash memory start address (0x8000) and its size can be changed by option byte. The boot code area includes the interrupt vector table (0x8000 - 0x8080), so to allow for modification of the vector table via IAP, the main vector table should be redirected to another vector table located in the rewritable application code area. See *Figure 2.* for memory map locations and dependencies.

Figure 2. User boot code area and User application area



More detailed information about memory map of particular STM8 device type can be found in STM8 Reference manual and datasheet.

1.4 Library support for Flash programming

The STM8 firmware library, available from st.com, provides developed and verified functions for programming Flash memory. Those functions supports byte/word and block programming. These library functions make it easier for developers to write their programming code. The library also includes functions for managing the programming code in RAM: copy to RAM, execution from RAM, storing functions in Flash memory.

The STM8 library package also contains examples that show how to use these functions in the final source code. You can write your own code based on these examples.

2 Library support for IAP

STM8 library contains modules for Flash programming. These are contained in the following files:

\FWLib\library\src\stm8s_flash.c

\FWLib\library\inc\stm8s_flash.h

\FWLib\library\inc\stm8s_map.h

Here you can find the complete source code for Flash programming. Refer to the library user manual, \FWLib\stm8s_fwlib_um.chm, for help on using STM8 library.

2.1 Flash programming function list

This list gives a short description of the STM8S Flash programming functions:

void FLASH_DeInit (void)
Deinitializes the FLASH peripheral registers to their default reset values.

void FLASH_EraseBlock (u16 BlockNum, FLASH_MemType_TypeDef MemType)
Erases a block in the program or data EEPROM memory.

void FLASH_EraseByte (u32 Address)
Erases one byte in the program or data EEPROM memory.

void FLASH_EraseOptionByte (u32 Address)
Erases an option byte.

u32 FLASH_GetBootSize (void)
Returns the Boot memory size in bytes.

FlagStatus FLASH_GetFlagStatus (FLASH_Flag_TypeDef FLASH_FLAG)
Checks whether the specified Flash flag is set or not.

FLASH_LPMode_TypeDef FLASH_GetLowPowerMode (void)
Returns the Flash behavior type in low power mode.

FLASH_ProgramTime_TypeDef FLASH_GetProgrammingTime (void)
Returns the fixed programming time.

void FLASH_ITConfig (FunctionalState NewState)
Enables or Disables the Flash interrupt mode.

void FLASH_Lock (FLASH_MemType_TypeDef MemType)
Locks the program or data EEPROM memory.

*void FLASH_ProgramBlock (u16 BlockNum, FLASH_MemType_TypeDef MemType, FLASH_ProgramMode_TypeDef ProgMode, u8 * Buffer)*
Programs a memory block.

void FLASH_ProgramByte (u32 Address, u8 Data)
Programs one byte in program or data EEPROM memory.

void FLASH_ProgramOptionByte (u32 Address, u8 Data)
Programs an option byte.

void FLASH_ProgramWord (u32 Address, u32 Data)
Programs one word (4 bytes) in program or data EEPROM memory.

u8 FLASH_ReadByte (u32 Address)
Reads any byte from Flash memory.

u16 FLASH_ReadOptionByte (u32 Address)
Reads one option byte.

void FLASH_SetLowPowerMode (FLASH_LPMode_TypeDef LPMode)
Select the Flash behavior in low power mode.

void FLASH_SetProgrammingTime (FLASH_ProgramTime_TypeDef ProgTime)
Sets the fixed programming time.

void FLASH_Unlock (FLASH_MemType_TypeDef MemType)
Unlocks the program or data EEPROM memory.

FLASH_Status_TypeDef FLASH_WaitForLastOperation (FLASH_MemType_TypeDef MemType)
Wait for a Flash operation to complete.

3 Compiler support (Cosmic) for RAM code execution

As mentioned above, block programming must be executed from RAM memory. Therefore the code to be copied into RAM must be compiled and linked to be run in RAM address space but stored in Flash memory.

It is possible to write simple programming code assembly, taking care with the RAM addressing and then storing this code in Flash (e.g. code uses only relative addressing or RAM addresses). However it is more efficient to use compiler support for this purpose. Cosmic compiler support (described below) has these features built-in.

3.1 Description how to compile/link code into RAM address locations

This option is performed (in Cosmic compiler) by creating a special memory segment which is defined in the linker file (*.lkf) and marked by flag *"-ic"* . For example (RAM space segments definition in linker file):

```
# Segment Ram:
+seg .data -b 0x100 -m 0x500 -n .data
+seg .bss -a .data -n .bss
+seg .E_W_ROUTs -a .bss -n bootcode -ic
```

This defines a RAM space from address 0x100. Firstly the .data and .bss sections are defined. Then it defines a moveable .E_W_ROUTs section where the routines for Flash memory erase and write operations will be located. This section must be marked by option *"-ic"* (moveable code).

Into *"-ic"* marked section we put functions which should be compiled/linked for RAM execution but stored in Flash memory. This is done in source code by section definition, for example:

```
...
//set code section to E_W_ROUTs placement
#pragma section (E_W_ROUTs)
void FlashWrite(void)
{
...
}
void FlashErase(void)
{
...
}

//set back code section to default placement
#pragma section ()
...
```

Now the *"FlashWrite()"* and *"FlashErase()"* functions will be compiled and linked for RAM execution (in section *"E_W_ROUTs"*) but their code will be placed in Flash memory - just after the *".text"* and *".init"* sections. This can seen from the generated map file:

Example of final map file:

```
                              --------
                              Segments
                              --------
start 00008080 end 00008500 length  1152 segment .text
start 00000000 end 00000000 length     0 segment .bsct
start 00000000 end 00000003 length     3 segment .ubsct
start 00000003 end 00000098 length   149 segment .RAM
start 00000098 end 00000098 length     0 segment .share
start 00000100 end 00000100 length     0 segment .data
start 00000100 end 00000100 length     0 segment .bss
start 00000100 end 000001F0 length   240 segment .E_W_ROUTs, initialized
start 00008510 end 00008600 length   240 segment .E_W_ROUTs, from
start 00008000 end 00008080 length   128 segment .const
start 00008500 end 00008510 length    16 segment .init
```

The map file shows the location of the *"E_W_ROUTs"* sections. One for storage in Flash (in map file marked as *"from"*) and one for execution (in map file marked as *"initialized"*).

Finally the microcontroller firmware must copy these sections from Flash to RAM before calling the RAM functions. Cosmic compiler support this copying by a built-in function *"int _fctcpy(char name)"* which copies a whole section from a source location in Flash to a destination location in RAM. For example:

```
void main(void)
{
    ...
    //copy programming routines to Flash
    _fctcpy('b');

    /* Define flash programming Time*/
    FLASH_SetProgrammingTime(FLASH_PROGRAMTIME_STANDARD);

    /* Unlock Program & Data memories */
    FLASH_Unlock(FLASH_MEMTYPE_DATA);
    FLASH_Unlock(FLASH_MEMTYPE_PROG);

    /* Fill the buffer in RAM */
    for (i = 0; i < FLASH_BLOCK_SIZE; i++) GBuffer[i] = new_val2;

    /* Program the block 0*/
    block = 0; /*block 0 is first block of Data memory: address is 4000*/
    FLASH_ProgramBlock(block, FL_MEMTYPE_PROG, FL_PRGMODE_STD, GBuffer);
...
}
```

As shown in the example above, just after the program starts, the RAM section is copied from Flash into RAM by the function *"_fctcpy('b')"*. The function parameter 'b' is the first character of the section name defined in the linker file (*"b"* as *"bootcode"* - see linker file *on page 8*) .

The firmware then can call any of the FLASH_xxx() functions in RAM.

Execution is then done from RAM.

4 STM8 devices with boot ROM - built-in IAP implementation

Some but not all STM8 devices have an internal Boot ROM memory which contains a bootloader and thus already have a built in IAP implementation. So the IAP example described in this application note is primarily intended for STM8 devices that have no Boot ROM.

4.1 Implementation details

The built-in bootloader in BootROM is fixed (not rewritable) and is specific for each STM8 type. The communication interface supported depends on the peripherals present in the given STM8 type and if they are implemented in the bootloader. For example, some device types support firmware download through UART and CAN, some device types support only UART and others only SPI. Information concerning the supported interfaces can be found in the relevant device datasheet.

Activation of the built-in bootloader is done by programming the BL[7:0] option byte described in the datasheet. The Boot ROM bootloader checks this option byte and if enabled then it runs it own code - waits for the host to send commands/data. If the BL[7:0] option byte is inactive then the Boot ROM bootloader jumps to user reset address - 0x8000.

4.2 Bootloader protocol

To be able to download firmware into the device, the host and bootloader must communicate through the same protocol. This bootloader protocol for STM8 devices is specified in the *"UM0560 - Bootloader user manual"* available from http://www.st.com. This same protocol is used in the firmware example provided with this application note. The UM0560 user manual describes all the bootloader protocol properties: used interfaces, timeouts, command formats, packet formats, error management, ...

5 Example of user bootloader application

This section describes a practical application of IAP programming with a user - implemented bootloader.

A bootloader is the part of code which runs immediately after a microcontroller reset and waits for an activation signal (for example from grounding a specific pin or receiving a token from a communications interface).

If activation is successful then the code enters bootloader mode. If activation fails (for example due to timeout, jumper on pin not present) then the bootloader jumps directly to the user application code.

In bootloader mode, the bootloader communicates with the external master device through a selected interface (UART, SPI, I2C, CAN, ...) using a set of commands. These commands usually are write to Flash, erase Flash, verify Flash and some additional operations: read memory, execute code from given address (jump to given address).

5.1 Bootloader firmware description

The following flow chart (*Figure 3.: Bootloader flowchart*) describes the basic program flow of the bootloader application.

After device reset, the bootloader checks if it is activated by predefined I/O pin. This pin works as bootloader activation signal and is connected via a jumper to ground. The bootloader configures the I/O pin in input mode and activates the pull-up on the I/O port. Then it checks the pin state. If the pin voltage level is zero (jumper present) then bootloader is activated. Otherwise it jumps to the reset address of the user application (if this address is valid).

If the bootloader is activated it then initializes the communication interface (UART1) in receiver mode and starts a timeout count (e.g. 1 second). If during this timeout nothing is received from UART1 then the bootloader jumps to the user application reset address (if valid).

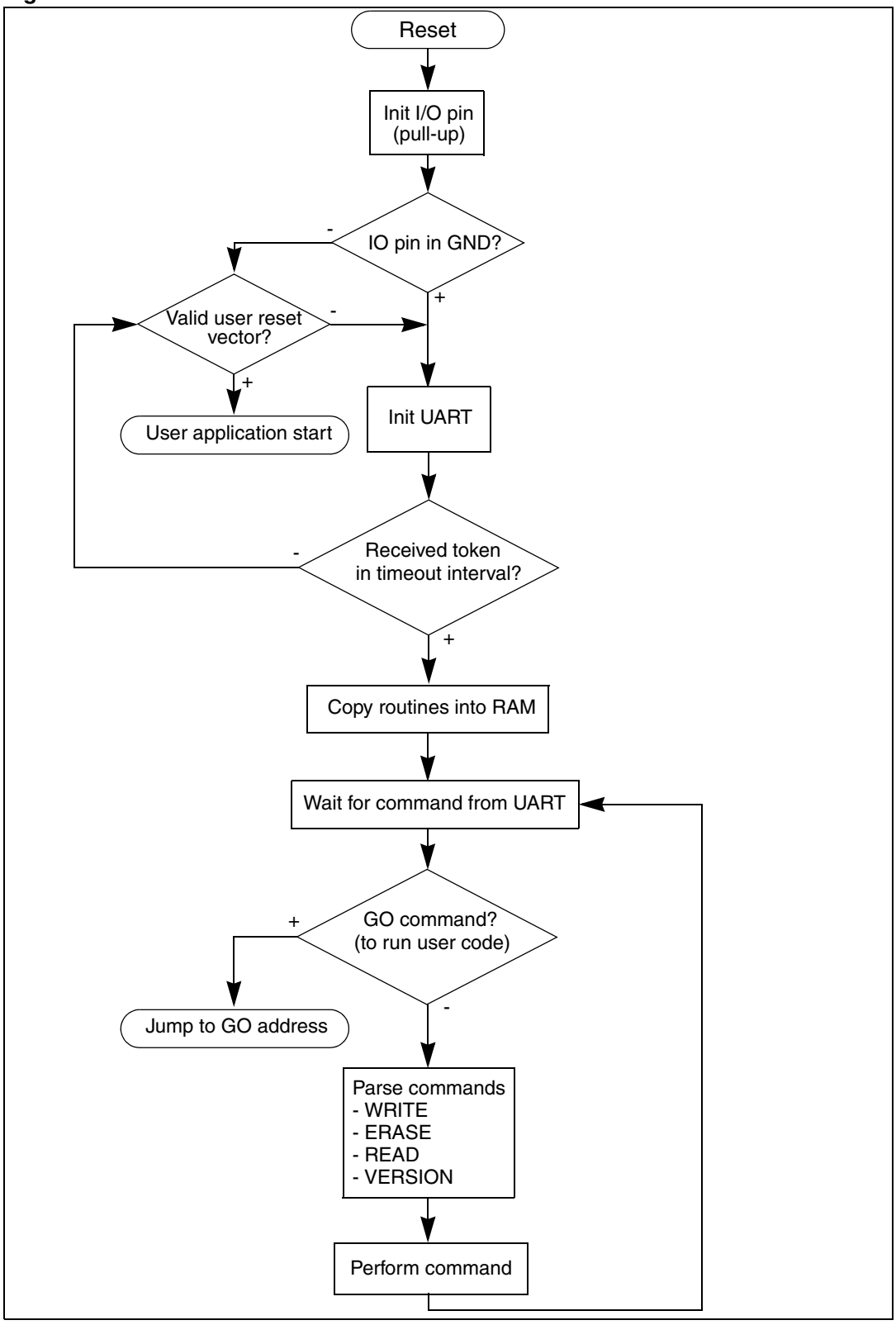
If the bootloader a valid activating token byte receives from UART1 before the timeout elapses, it then enters memory management mode to perform the following operations:

1. First the bootloader initializes the Flash programming routines by copying the programming functions to RAM.
2. Then bootloader waits in a loop for a valid command to be received from UART1.
3. If valid command is received it is parsed and executed (Read, Write, Erase, Version commands).
4. If a Go command is received, then the bootloader jumps to the address given in the command (from which can be returned or not depending on the code at the destination address).

Commands have given format which is specified and must be followed by both sides: bootloader and master. The command specification handles all possible cases and covers error management. To make it easy to meet this specification while performing the more complicated operations (for example, fill Flash memory from “*.s19” file), appropriate software with a graphical user interface (GUI) has been written for the master side (usually

PC). This GUI can perform all the specified commands. The command specification is included with this Application Note as separate document.

Figure 3. Bootloader flowchart



5.2 Used library functions

The code available with this application note uses STM8 software library functions by including the appropriate files into projects. It uses not only the functions for Flash memory programming management but also the functions for timer management (for example for timeout counting), UART management (for communications with the master), GPIO management (for configuring the I/O pin), clock management (to set up microcontroller and peripheral speed).

The use of the library simplifies the development and makes the main source code simpler.

5.3 Block programming - RAM code copy

Copying the RAM code from Flash is done with Cosmic compiler support - by the built-in function *"_fctcpy()"*. Copying is performed immediately after microcontroller reset - at the beginning of the *"main()"* program function.

If another compiler is used, this would require a similar alternative solution for building RAM routines.

5.4 Interrupt vector table redirection

The primary interrupt vector table is located in the UBC (User Boot Code) area which is write protected. Interrupt table redirection has been implemented to allow the user application to change the interrupt vectors.

Redirection is performed in the following way. The start of the user application area contains its own interrupt table with the same format as the primary interrupt table, a set of jumps to interrupt routines (the first table entry is the "user application reset"). The primary interrupt table contains fixed vectors to this user interrupt table. So if an interrupt occurs then it is redirected from the primary interrupt table to the user interrupt table by one jump.

The only requirement for the user application is that the user interrupt table must be located at a fixed address because the primary interrupt table is not rewritable and jumps to the user interrupt table at a fixed address). See *Figure 2.: User boot code area and User application area* for details.

5.5 Bootloader overwrite protection

The bootloader area must be write protected. This protection is done by setting the UBC option byte according to the bootloader size.

The user application area (which begins with the user interrupt vector table) starts in first unprotected Flash sector after the UBC area.

Bootloader protection must be set by SWIM - usually during bootloader programming.

5.6 Used bootloader interface

The current version of the bootloader code supports UART as communication interface. In future versions, other communication interfaces will be supported (SPI, I2C, CAN, LIN). The

user application code then can be upgraded via the interface used by the target application (for example, CAN in the case of an automotive system).

6 Conclusion

The STM8 microcontroller has full support for IAP programming. It allows the internal STM8 firmware to erase and write to any of its internal memories: Flash program memory, data EEPROM memory, RAM memory. The short programming times (inherent to the advanced technology) can be further decreased by using of fast programming mode (if the destination memory is erased). The STM8 RWW (Read While Write) feature allows the microcontroller to stay at a high performance level even during Flash and/or EEPROM memory programming.

The bootloader application presented here, provides designers with an IAP implementation that they can use to add IAP support into their own final products. The bootloader code is designed to run in STM8 family microcontrollers and uses the STM8 software library for easier programming.

6.1 Features in final bootloader application

- Bootloader activated by external pin (jumper on PCB)
- Interrupt table redirection
- Executable RAM code management
- Library support for easier programming
- Read While Write feature
- High level C-language usage
- In future: support for multiple communication interfaces

7 Revision history

Table 1. Document revision history

Date	Revision	Changes
20-Feb-2009	1	Initial release