AN1576
APPLICATION NOTE

IN-APPLICATION PROGRAMMING (IAP) DRIVERS FOR ST7 HDFLASH OR XFLASH MCUs

by Microcontroller Division Applications

INTRODUCTION

The In-Application Programming (IAP) architecture defined by STMicroelectronics gives a large flexibility in terms of the communication method used to (re)program a ST7 FLASH Microcontroller on board: not only the physical channel (I/Os, SPI, UART, USB, CAN,..) for receiving the new data, but also the protocol (Commands, Status, Data structure,..) can be user specific.

The principle of the IAP process (See Application Note AN1575 for further details) is to execute from a protected memory area, Flash Sector 0, a firmware module that reprograms the remaining memory area: In order to help you develop you own reprogramming firmware, STMicroelectronics provides generic IAP drivers that can be used whatever the protocol or physical layer.

This application note presents these two generic In-Application Programming drivers: one for HDFLASH based MCUs and one for XFLASH based devices.

The architecture and the software interface as well as some practical examples are presented for each of these two drivers.

1 XFLASH DRIVER

The XFlash driver is a software which can be compiled using either C-Metrowerks or C-COSMIC and supports all memory models.

The driver is composed of 2 files: XFlash.c and XFlash.h to be included in the project. It provides RASS_Disable, and XFlashWriteBlock functions. No *Sector Erase* function is provided, since it can be emulated by writing 0xFF in a memory block.

The driver code is approximately 256 bytes in size, this may vary slightly depending on the compilation.

1.1 DRIVER FUNCTIONS

1.1.1 RASS_Disable

```
void RASS_Disable(unsigned char key1,unsigned char key2)
```

This function disables the RASS protection to allow write access to XFLASH sectors 1 & 2, if the two user variables key1 and key2 have the correct values: 0x56 and 0xAE.

To increase the reliability of the system, the software sequence which writes the hardware keys should not be stored in the program memory. Both hardware keys must always be loaded externally (via I/O ports, SCI, etc...). This security feature prevents any wrongful access after a program counter corruption.

1.1.2 XFlashWriteBlock

```
unsigned char XFlashWriteBlock(unsigned char *Buffer, unsigned int Flash, unsigned char ByteNb)
```

This function performs write operations with the following parameters:

– *Buffer* points to the start address of the new data to be programmed

– *Flash* points to the XFLASH start address area where the new data is to be programmed

– *ByteNb* is the number of bytes to be programmed

The ByteNb can take a value of up to 256 even though the XFLASH itself is programmed in rows of 32 bytes max: The XFLASH driver handles the iterative loops automatically in cases where ByteNb is greater than 32.

The function returns a status: 0 means a failed operation, while a 1 means a successful operation.

1.1.3 User functions

The XFlash driver allows user specific functions to be executed while an XFlashWriteBlock function is executing. Writing 256 bytes requires several tenths of a millisecond during which time you may have to manage some tasks: A typical example is to refresh the watchdog, in order to avoid a Reset during XFlash reprogramming.

For this purpose the driver provides 1 function named UserWhileWriteBlock that you have to define in your firmware (take care to not use the RAM area with the driver variables).

Finally, the Interrupts are reenabled (RIM assembler instruction) by the driver before the execution of the UserWhileWrite function.

1.2 DRIVER ARCHITECTURE

1.2.1 Driver principle

When called by the user firmware in Flash Sector 0, the XflashWriteBlock executes the following sequence:

– Load the RAM Page 0 with a programming algorithm

– Execute this programming algorithm from RAM

– Return to the user firmware in Flash Sector 0, just after the XflashWriteBlock instruction.

At this point the RAM area used by the programming algorithm is released and is available to the user firmware.

1.2.2 Memory management

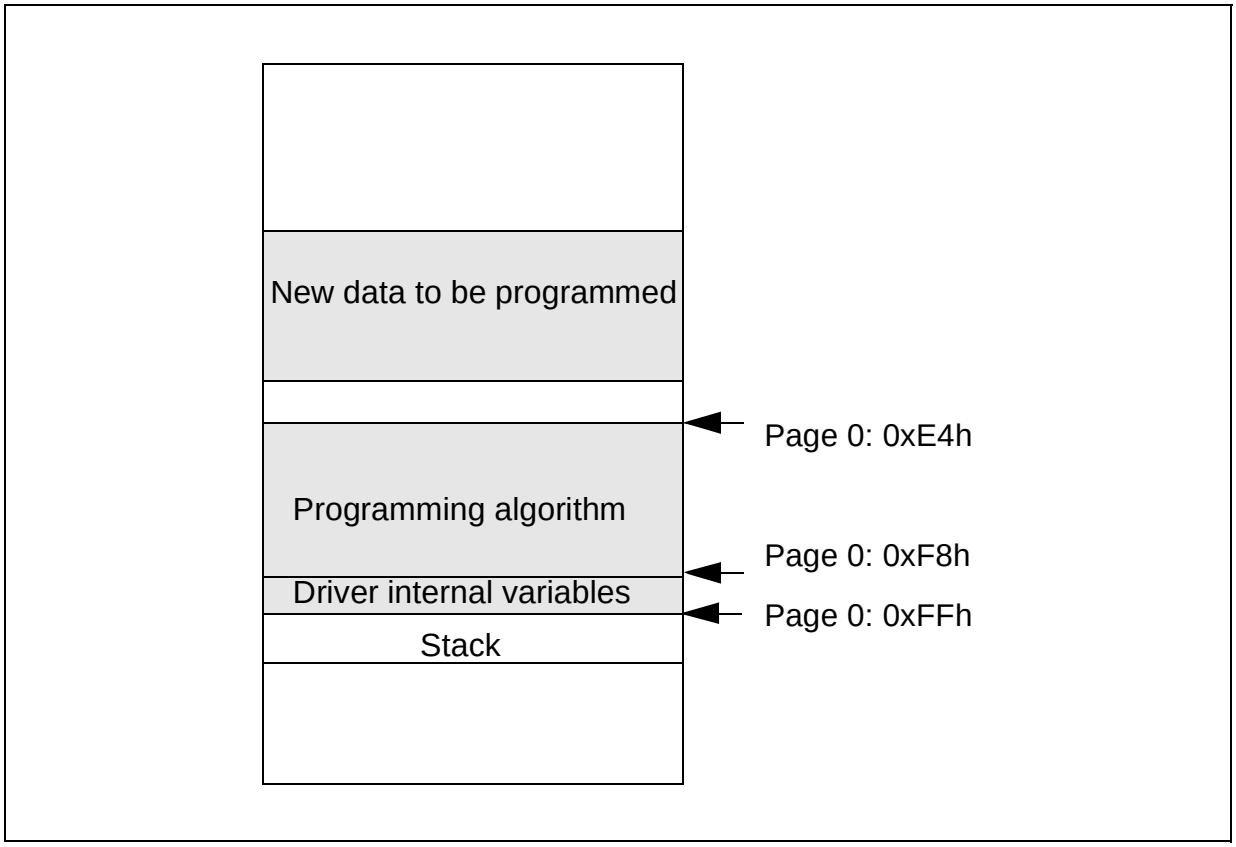
The programming algorithm requires 20 bytes and is positioned at the end of RAM Page 0.

In addition, 7 bytes in RAM Page 0 are reserved for the IAP driver internal variables.

The location of the 27 bytes (algorithm plus variables) is defined in the Xflash.h file by the parameter STACK_END: The location is then 0xE4h-0xFFh for a ST72F264 MCU.

It is important to note these 27 bytes are used only during the XFlashWriteBlock execution: You can still use this area in his application firmware (except in the UserWhileWriteBlock function) as long as he accepts they are overwritten during the IAP process.

Figure 1. RAM usage with XFLASH driver: ST72F264 example



1.2.3 Interrupts

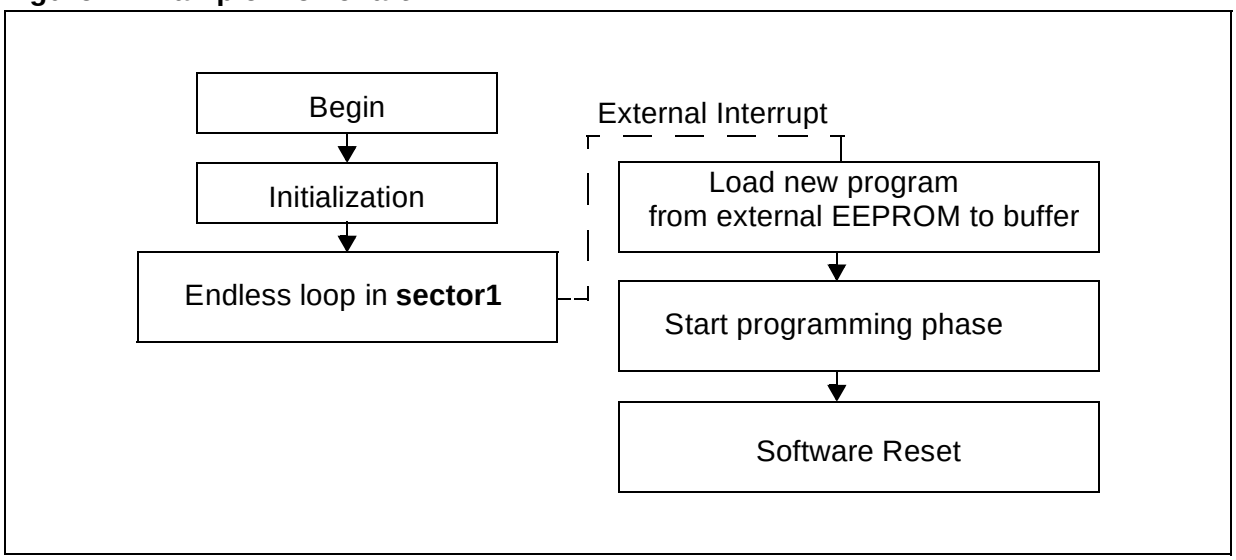
All the interrupt sources are disabled (SIM assembler instruction) by the XFlash driver.

1.3 IMPLEMENTATION GUIDELINES

An example is used to describe some generic guidelines you have to follow when developing an application with IAP capability. It includes all the modules you need for an application with IAP:

- Reset and initialization routines
- Basic user interface
- Reprogramming routines (On external Interrupt)
- Reprogrammable software module in sector 1

Figure 2. Example Flowchart



1.3.1 Project Configuration

1.3.1.1 Driver Configuration

The first step is to correctly configure the stack position in the Xflash.h file. This parameter is defined by the variable STACK_END. The XFlash driver uses this value to address its internal variables located at the top of RAM Page 0.

1.3.1.2 Program Memory

Special attention must be paid to the fact that some routines have to be executed from Sector 0 to ensure reliable operations.

Therefore pragmas have to be defined in such a way that Sector 0 includes: the XFlash driver, the Reset, Initialization and UserWhileWriteBlock routines as well as any library routines (Delay loops,..) needed for the reprogramming process.

1.3.2 RAM Management

1.3.2.1 New Code Buffer

It is recommended to define the buffer with the new code as a local variable at a fixed address in RAM Page 0. This has two advantages:

– The memory area is released by the IAP process.

– There is no risk of data overlap with the other local variables handled by the compiler.

Finally, its size must be defined in order to not conflict with the XFLASH driver and its variables: Its maximum size is then 128-27= 101 bytes. In this example, a 64 byte size has been chosen: The 128 bytes are programmed in two steps of 64.

1.3.2.2 Stack

One advantage of the XFLASH driver is that it does not use the stack: The XFlashWriteBlock function can be called by the firmware without any special precautions, and uses local variables. Therefore a local loop can be used to manage data ranges over the buffer size, as shown in the example given in Section 1.3.3 .

1.3.3 Firmware template

```
#include "io72264.h"/* ST72264 memory and register mapping */
#include "Xflash.h"   /* XFlash driver software        */
void Loop(void);
/************************Main loop ************************************/
void main(void)
{
/************************ Initialization *********************************/
ST7_Init();         //General initialization
while(1) Loop();
}
/*****************UserWhileWriteBlock************************************/
void UserWhileWriteBlock(void)
{
WDGCR=0x08;             //Refresh WATCHDOG Timer
}
/***************Programming routine within External ISR*******************/
@interrupt void On_DEMO(void)
{
unsigned char tmp;
unsigned char RamBuffer [64]@0xA6; /* New code and RASS keys buffer */
SPI_Init();               //initialize SPI communication
SPI_Rx(RamBuffer,2,100,(char)0);     //Receive RASS keys from EEPROM
RASS_Disable(RamBuffer[0],RamBuffer[1]); //Unlock Flash
for (tmp=0;tmp<2;tmp++)
{
SPI_Rx(RamBuffer,64,64*tmp,(char)0);
XFlashWriteBlock(RamBuffer,(unsigned int)&Loop+64*tmp,64);
}
SPI_Disable();
WDGCR=0xC0; //Resets the MCU which also relocks the XFLASH;
}
/************************Sector 1 software*******************************/
#pragma section (Loop)
void Loop(void)
{
}
```

2 HDFLASH DRIVER

The HDFlash driver software can be compiled using C-Metrowerks or C-COSMIC.

The driver is composed of 2 files: HDFlash.c and HDFlash.h to be included in the project. In its minimum configuration, it provides RASS_Disable, HDFlashWriteBlock and HDFlashErase-Sector functions. In this configuration, the driver code is approximately 85 bytes in size, this may vary slightly depending on the compilation.

However, the HDFlash.c file includes some other functions you can decide (#define) to use.

2.1 DRIVER FUNCTIONS

2.1.1 RASS_Disable

`void RASS_Disable(unsigned char key1,unsigned char key2)`

This function disables the RASS protection to allow write access to HDFLASH Sectors 1 & 2, if the two user variables key1 and key2 have the correct values: 0x56 and 0xAE.

To increase the reliability of the system, the software sequence which writes the hardware keys must not be stored in program memory. Both hardware keys must always be loaded externally (via I/O ports, SCI, etc...). This security feature prevents any wrongful access after a program counter corruption.

2.1.2 HDFlashWriteBlock

`HDFlashWriteBlock(unsigned char *Buffer, unsigned char *Flash, unsigned char ByteNb,unsigned char Freq)`

This function performs write operations with the following parameters:

– *Buffer* points to the start address of the new data to be programmed

– *Flash* points to the HDFLASH start address area where the new data is to be programmed

– *ByteNb* is the number of bytes to be programmed

– *Freq* defines the programming pulse

The ByteNb can take a value of up 256 even though the HDFLASH itself is programmed byte by byte: The HDFLASH driver handles the iterative loops in cases where ByteNb is greater than 1.

This function returns a value 01 when successful

2.1.3 HDFlashEraseSector

`HDFlashEraseSector(unsigned char Sector, unsigned char Freq)`

This function performs the erase operations with the following parameters:

Sector is the sector number (1 or 2).

Freq is the internal frequency of the Microcontroller

This function returns the value 03 when successful.

2.1.4 User functions

The HDFlash driver allows user specific functions to be executed during a HDFlashWriteBlock or HDFlashEraseSector function. This these operations require several tenths of a millisecond in during which you may have to manage some other tasks: A typical example is to refresh the watchdog, in order to prevent a Reset while the HDFlash is being reprogrammed or erased.

For this purpose the driver provides 2 functions named UserWhileWriteBlock and UserWhileErase, that you have to define in your firmware.

2.1.5 Programming/Erasing Supply Voltage Management

The HDFlash driver does not take into account the management of the external VPP supply. This is because the VPP implementation is totally user specific and cannot be covered by a generic driver.

2.2 DRIVER ARCHITECTURE

2.2.1 Driver principle

The HDFlashWriteBlock function works as follows:

Once called by the user firmware in sector0, HDFlashWriteBlock and HDFlashEraseSector execute the following sequence:

– Load the RAM memory at the current stack pointer with the programming algorithm

– Execute this programming algorithm from the RAM

– Return to the user firmware in sector 0, just after the HDFlashWriteBlock or HDFlashEraseSector instruction.

At that point the RAM area used by the programming algorithm is released and is available to the user firmware.

Note that the stack pointer remains at the value before the HDFlashWriteBlock or HDFlashEraseSector call.

2.2.2 Memory management

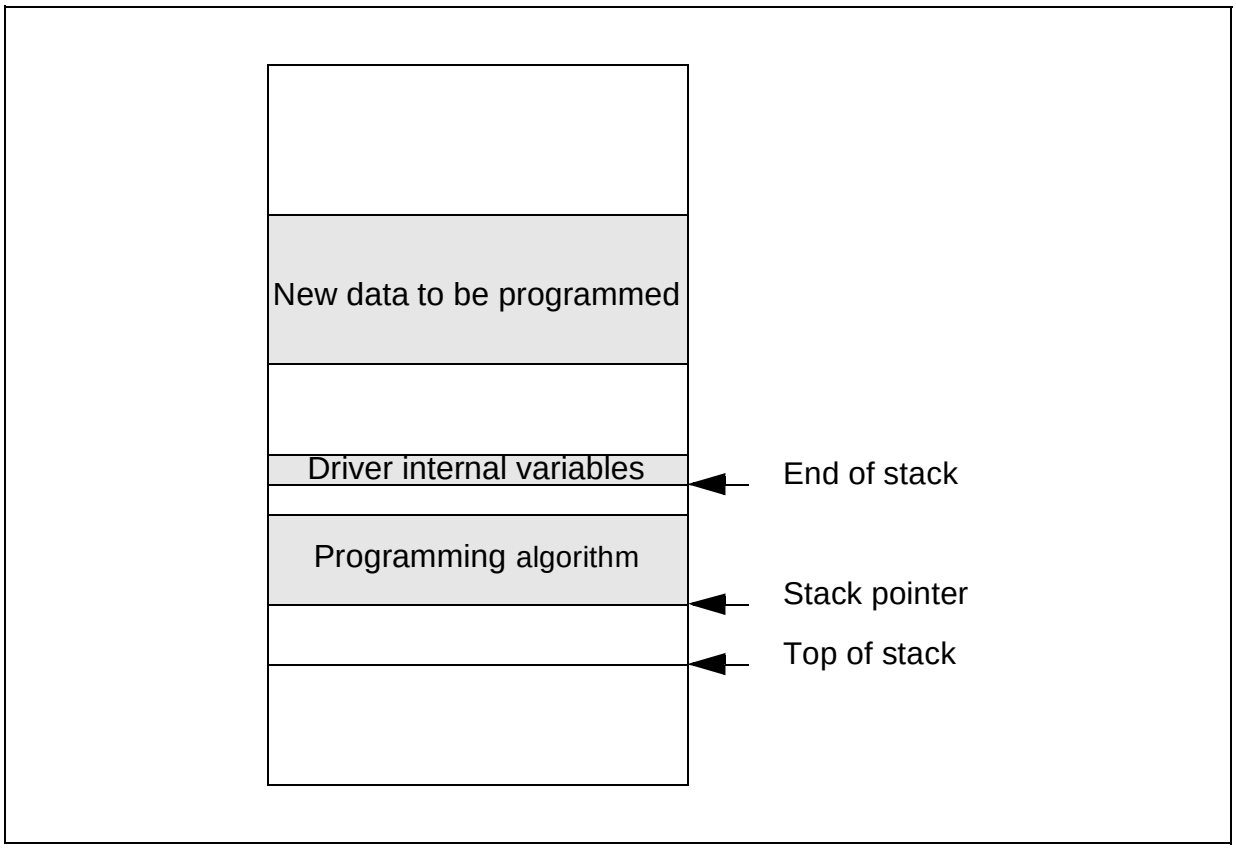
The programming algorithm requires 124 bytes and is positioned at the current stack pointer: Part of it can then be loaded into RAM Page 0 (Adjacent to the stack) in case the stack is already filled.

Take care that you have at least 124 bytes free in the stack when calling the HDFlashWriteBlock functions. This condition can be satisfied by reducing the number of stacked “Calls” between the main loop and the HDFlashWriteBlock or HDFlashEraseSector function.

In addition, 16 bytes in RAM Page 0 (Sort addressing mode) are reserved for the IAP driver internal variables. Their location is 0xF0-0xFF and cannot be changed.

Note: These variables are used only during HDFlashWriteBlock or HDFlashEraseSector execution: you can still use this area in your application firmware as long as you accept they may be overwritten during the IAP process.

Figure 3. RAM usage with HDFLASH driver

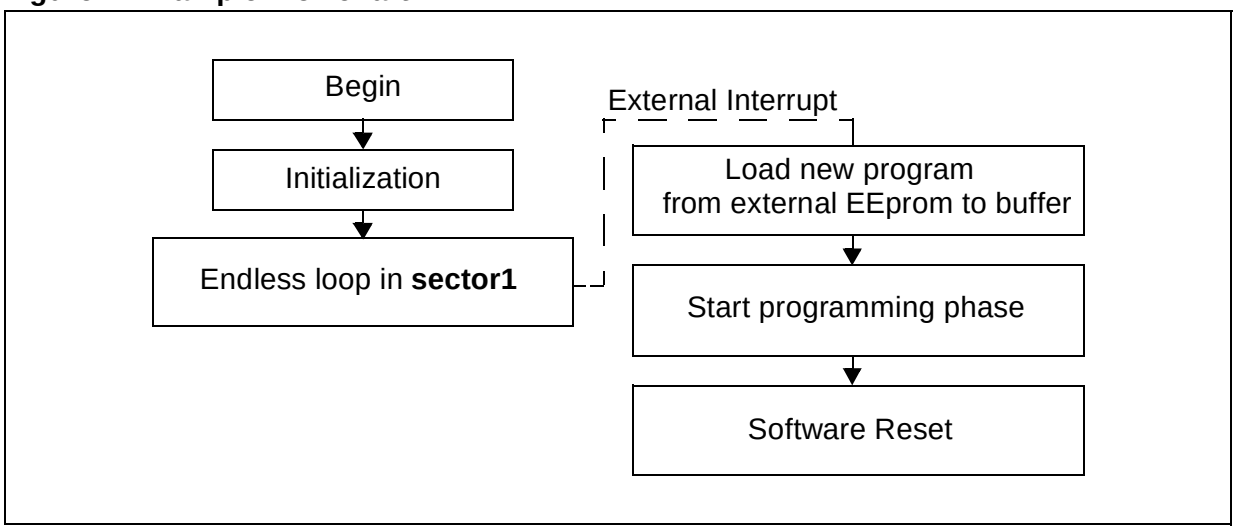


2.3 IMPLEMENTATION GUIDELINES

An example is used to describe some generic guidelines you should follow when developing an application with IAP capability. It includes all the modules you need or an application with IAP:

- Reset and initialization routines
- Basic user interface
- Erasing & Reprogramming routines (On external Interrupt)
- Reprogrammable software module in sector 1

Figure 4. Example Flowchart



2.3.1 Project Configuration

2.3.1.1 Driver Configuration

There is no need for driver configuration: Each MCU has an embedded programming algorithm already configured.

2.3.1.2 Program Memory

Be very careful that some routines have to be executed from Sector 0 to ensure reliable operations

Therefore pragmas have to be defined in such a way that Sector 0 includes: the HDFlash driver, the Reset, Initialization, UserWhileErase and UserWhileWriteBlock routines as well as any library routines (Delay loops,..) needed for the reprogramming process.

2.3.2 RAM Management

2.3.2.1 New Code Buffer

The buffer with the new code must be defined outside the stack, and outside of the F0h-FFh range.

In addtition, it is recommended to define it as a local variable at a fixed address, which presents a double advantage:

The memory area is released by the IAP process.

There is no risk of data overlap with the other local variables handled by the compiler.

2.3.2.2 Stack

As already stated, the HDFLASH driver occupies 124 bytes in the stack. This generally leaves only 4 bytes free in RAM and limits the possibility of using local function calls or variables during the programming or erasing process. A workaround consists of creating a macro equivalent to the HDFlashWriteBlock or HDFlashEraseSector functions: Using a macro instead of a function call saves 2 bytes in the stack.

For the same reason, it is highly recommended to limit the stack usage in the UserWhileWriteBlock and UserWhileErase functions.

2.3.3 VPP Supply

Two specific functions must be available in sector 0, in order to enable or disable the external 12V VPP supply during the programming or erasing operations.

In case of some supply generators, such as charge pump, a stabilization delay has to be included in the firmware.

2.3.4 Firmware template

```
#include "io72521.h"/* ST72521 memory and registers mapping */
#include "HDFlash_macros.h"  /* HDFlash driver software    */
unsigned char tempo; /*stabilization delay used by macro */
/********VPP Supply 1ms stabilization delay*******************/
#define Vpp_Enable()
{ \
 ClrBit(PADR,1); \
 for (tempo=0;tempo<1000;tempo++);
}
#define Vpp_Disable() SetBit(PADR,1) /* Disable Vpp macro     */
void Loop(void);

/*************** Main*************************************/
void main(void)
{
/******************* Initialization ***********************/
 ST7_Init();              //General initialization
while(1) Loop();
}
/*****************UserWhileWriteBlock************************
void UserWhileWriteBlock(void)
{
WDGCR=0x08;         //Refresh WATCHDOG Timer
}
/*************** UserWhileErase ****************************
void UserWhileErase(void)
{
WDGCR=0x08;         //Refresh WATCHDOG Timer
}
/****Erasing & Reprogramming routines within External ISR*******/
@interrupt void On_SW1(void)
{
 unsigned int cpt;
unsigned char *pFlash;
unsigned char ProgBuffer[45] @0x9C;
SPI_Init();              //initialize SPI communication

SPI_Rx(ProgBuffer,2,100,(char)0);     //Receive RASS keys from EEPROM
RASS_Disable(ProgBuffer[0],ProgBuffer[1]); //Unlock Flash

Vpp_Enable();
HDFlashEraseSector(1,1); //Erase Secor 1
for(cpt=0;cpt<2;cpt++)
```

```
{
SPI_Rx(ProgBuffer,45,45*cpt,(char)0); // Get data from EEprom
pFlash = (unsigned char*)(0xE000+45*cpt); // update flash pointer
HDFlashWriteBlock(ProgBuffer,(unsigned int )pFlash, 45, 1); //write first block
}
Vpp_Disable();
SPI_Disable();
WDGCR=0xC0; //Update watchdog
}
/**************************Sector 1 software*******************************/
//Section defined in sector 1 in lkf file
#pragma section (Loop)
void Loop(void)
{
//Nothing to be done at first Reset
}
******* STMicroelecetronics 2002 ********************* End Of File ************/
}
```

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 CONNECTION WITH THEIR PRODUCTS.