NEC

Preliminary User's Manual

µPD78F0828B

8-bit Single-Chip Microcontroller

Flash Self-Programming Library V 1.0

Document No. U14995EE1V0UM00
Date Published November 2000

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

The related documents in this publication may include preliminary versions. However, preliminary versions are not marked as such.

The export of this product from Japan is regulated by the Japanese government. To export this product may be prohibited without governmental license, the need for which must be judged by th customer. The export or re-export of this product from a country other than Japan may also be prohibited without a license from that country. Please call an NEC sales representative.

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.l.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Vélizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

NEC Electronics (France) S.A.
Spain Office
Madrid, Spain
Tel: 91-504-2787
Fax: 91-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore 1130
Tel: 65-253-8311
Fax: 65-250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC do Brasil S.A.
Electron Devices Division
Rodovia Presidente Dutra, Km 214
07210-902-Guarulhos-SP Brasil
Tel: 55-11-6465-6810
Fax: 55-11-6465-6829

J99.1

Preface

Readers This manual is intended for users who want to understand the functions of the µPD78F0828B.

Purpose This manual presents the hardware manual of µPD78F0828 .

Organization This system specification describes the following sections:

- Pin function
- CPU function
- Internal peripheral function
- Flash memory

Legend Symbols and notation are used as follows:

Weight in data notation	:	Left is high-order column, right is low order column
Active low notation	:	$\overline{xxx}$ (pin or signal name is over-scored) or /xxx (slash before signal name)
Memory map address:	:	High order at high stage and low order at low stage
Note	:	Explanation of (Note) in the text
Caution	:	Item deserving extra attention
Remark	:	Supplementary explanation to the text
Numeric notation	:	Binary . . . xxxx or xxxB Decimal . . . xxxx Hexadecimal . . . xxxxH or 0x xxxx

Prefixes representing powers of 2 (address space, memory capacity)

k (kilo) : 2^{10} = 1024

M (mega) : 2^{20} = 1024^2 = 1.048.576

G (giga) : 2^{30} = 1024^3 = 1.073.741.824

[MEMO]

Table of Contents

List of Figures

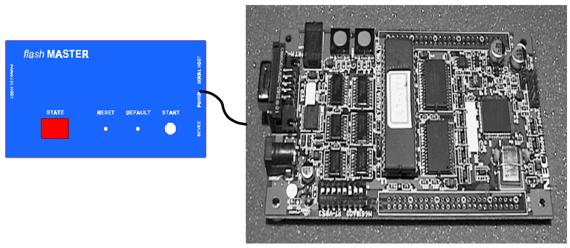
Chapter 1 Introduction

1.1 General topics

This users manual shall serve as a programming-aid for any customer who is using the NEC-K0-family Secure SelfProgramming-firmware with Realtime-support like it is implemented in the µPD78F0828B (CANASSP3). Both, Hard- and Software-Environment have to be configured to use Self-Programming. Two different modes are possible to access the Flash-Memory:

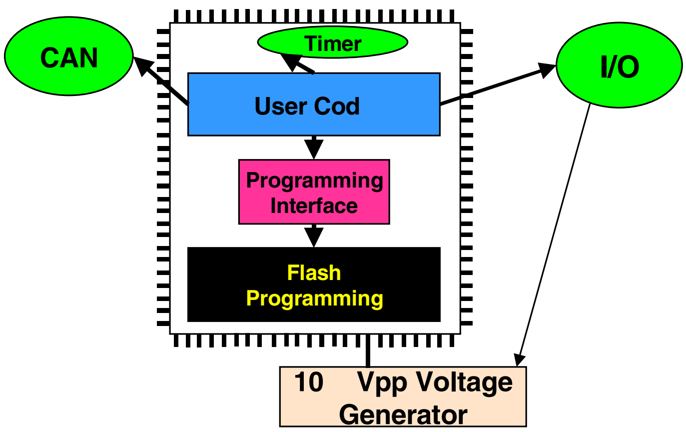
- **Using the OnBoard-Programming Mode:** An initial program has to be programmed into the device via a programming-tool like the FLASH-Master. It can be connected to an interface on an application-target board. This program is either the full code to be programmed, or should include a so called BOOT-LOADER located in the lower FLASH-memory-area. The main task of this BOOT-LOADER is to start basic software-functions after leaving the RESET-mode and to support the Flash-Self-Programming-functionality for reprogramming from a connected network.

Figure 1-1: Hardware-environment of Flash-Master



- **Using the Self-Programming Mode:** In self-programming mode data can be programmed to the flash. Data can be parameters or program code that got into the device by support of the user program. Therefore the user program can use all device specific peripherals like CAN, UART and timers. Once the data is in the device, e.g. buffered in the RAM area, it can be programmed via an internal programming interface to the upper half flash area. The flash-ROM is divided into two blocks, an upper (block 1) and a lower block (block 0). Only the upper block can be programmed in self-programming mode.To write data also into the lower block, the two blocks can be switched with the swap self-programming function, so that the old lower block will be permanently the upper block. This new constellation is also valid after RESET. A further call of the swap routine will fail, until the present upper block is erased.

Figure 1-2: Layout with Self-Programming



This document explains how self-programming is working and how it can be used under assistance of an API (Application Programming Interface). This API is realized as a library, written in a higher-level programming language (ANSI-C). It should facilitate the use of the self-programming functions out of the user program.
In the second chapter the environment is described how this API can be used in programs to get full self-programming functionality.
This document is based on the device 78F0828B but can easily be used for other devices with the same self-programming technology and real time support.

1.2 Self-Programming on µPD78F0828B

The following chapter will describe the main-topics of Self-Programming on the µPD78F0828B and how the programming is done on the chip.

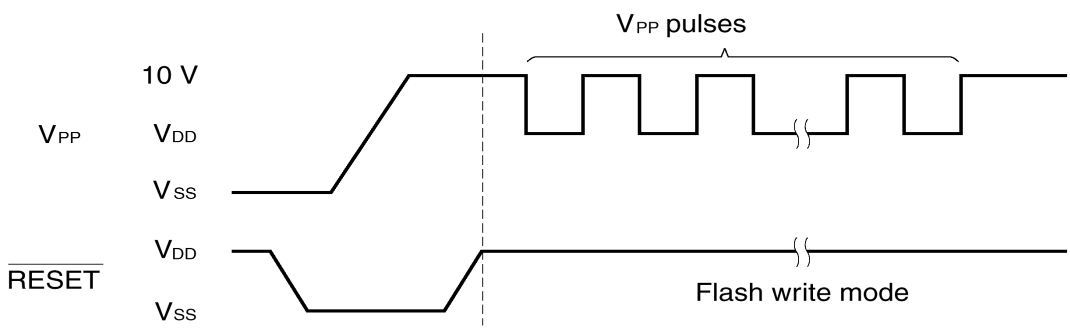
1.2.1 Description

Starting with a un-programmed chip, the user is responsible to program a boot-loader, supporting the Self-Programming-functions, in the lower block of the Flash-Area. The Self-Programming support- functions must be included in the final application program, too, if data- or parameter-updates are further necessary.
By a user-defined communication-channel data must be written to a specified RAM-Area; this data is later-on programmed to the upper-memory-block.
After programming of upper-ROM-block is done, the ROM-blocks are swapped so the actual code resides in the old ROM-block1. Now, the new ROM-block0 can be programmed, whereby all Flash-Memory is Write-accessible.

1.2.2 Control of V_{PP}

- Normal Flash-Programming: The user can program the Flash-Area, but no security and no support of RAM-based functions are given while running the Firmware-routines
- Secure Flash Self-Programming: The user can program the Flash-Area, but in case of failures, at least one ROM-block is recognized to contain valid data.
- RTSF-Selfprogramming: The user can run any function without interrupt, but using watchdog, out of a pre-defined RAM-Area while using the Self-Programming routines. This is called RealTime-SupportFunction.

Figure 1-3: Outline of Programming-Voltage



While RESET release different modes can be selected:

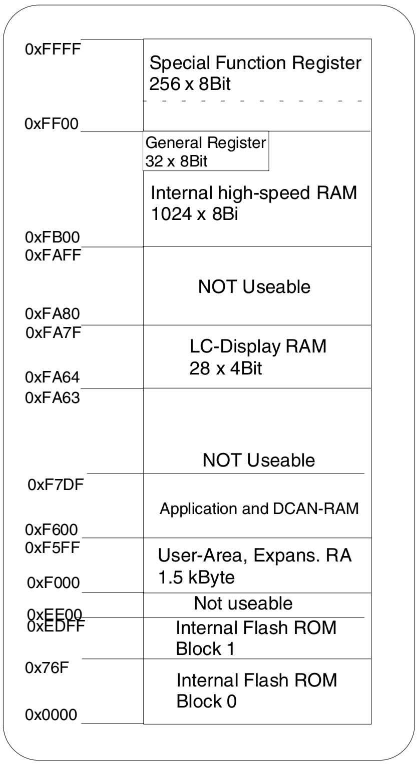
- Flash Mode: Connected 10 V to VPP at rising RESET signal will switch to the flash programming mode. The used serial communication interface is selected by pulsing the VPP voltage. The principle is shown in figure 1-3.
- Normal mode: Ground level is attached to Pin 14; the chip will start user program execution at the address defined by the RESET vector at address 0000h.
- Self-programming mode: Self-programming mode is called out of the normal mode after 10 V supply to VPP. To achieve a secure programming during all events, like e.g. power fail, the lower flash block can not be programmed. To program the whole flash, both flash blocks can be exchanged once. For additional exchanges the upper block has to be erased first.

Real time functions like watch-dog or network communication are also supported during self-programming with the RTSF (Real Time Support Function). RTSF is user code located in RAM and is automatically called within 1ms out of the Self-programming mode.

1.2.3 Memory-map in Normal-Mode

In the normal-Mode, the memory-configuration looks like follows:

Figure 1-4: Normal-mode memory-layout

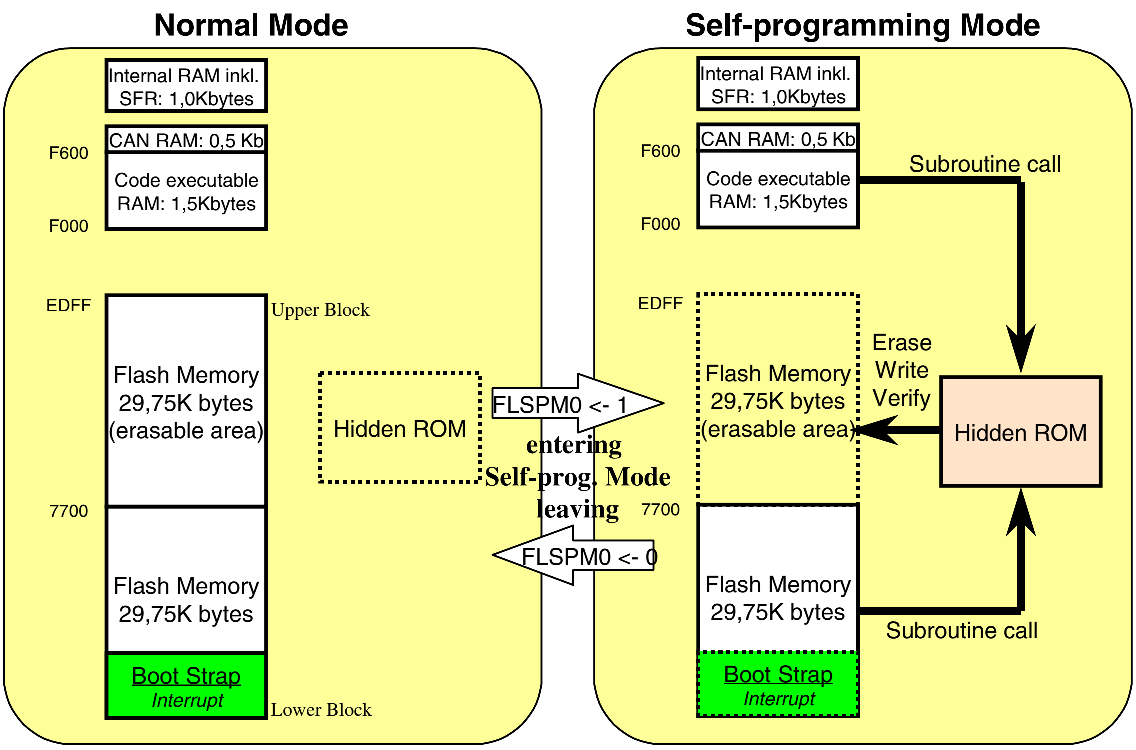


As described, the ROM is divided into an upper and a lower block; only the upper block is directly accessible for the Self-Programming routines. Nevertheless, the whole Flash-Area is programmable by using the Self-Programming Swap-Function.

1.2.4 Entering Self-Programming Mode

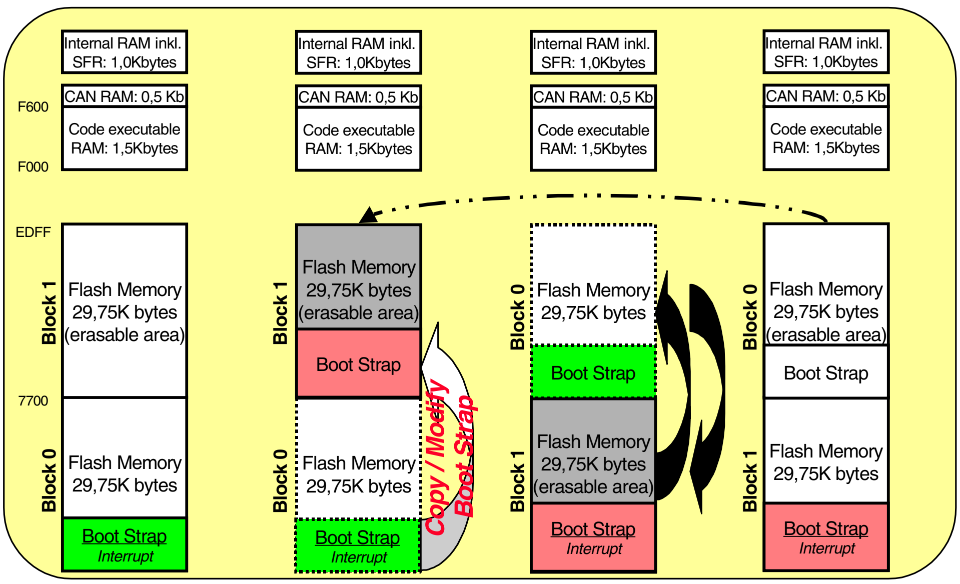
After applying the 10 V to VPP and entering Self-Programming mode, the upper ROM-block will be switched with the hidden-ROM-Area.

Figure 1-5: Memory-configuration after entering SelfProgramming-Mode



From this Hidden-ROM-area, all Self-Programming-functions calls are executed. After the boot-strap and the code is written, the upper ROM-block contains the new program, so the ROM-blocks are re-switched to enable the new code. Now the new block1 can be programmed. The below figure shows this in detail.

Figure 1-6: Detailed description of Flash-Programming



1.2.5 The Entry-RAM

The Entry-RAM-Area is a pre-defined area with a length of 32 bytes. These bytes contain all data necessary for the firmware-routines and are initialized by the library-call-function SP_Init.
Normally, pre-defined values are stored here by the Init-Function, so the user shouldn't change them. If there is a need to change this values, please follow the information provided in this chapter.

Following picture shows the structure of the EntryRAM-Area.

```
struct ERAM
{
  unsigned char _Reserved1;
  unsigned char _Reserved2;
  unsigned int  _FlashMemStartAdr;
  unsigned int  _Reserved3;
  unsigned char _ByteInFlash_BlockNo;
  unsigned char _WriteTimeData1;
  unsigned char _EraseTimeData1;
  unsigned char _EraseTimeData2;
  unsigned char _EraseTimeData3;
  unsigned char _ConvTimeData1;
  unsigned char _ConvTimeData2;
  unsigned char _ConvTimeData3;
  unsigned int  _WrDatStorStartAdr;
  unsigned int  _RAMRoutStartAdr;
  unsigned int  _FlashSwapRetAdr;
  unsigned char _IntervTimeData1;
  unsigned char _Reserved4[11];
};
```

The start-address of the entry-RAM-Area is defined in the API-header file and respective the linker-file. The important elements in the Entry-RAM are described as follows:

- FlashMemory Start-Address: Here the address takes place, where the Flash-Writing routine starts to write

- No. of bytes written in flash-memory: This value has to been set to 0x01allways - description of the ROM-block number where firmware-routines take effect; only the Write-function is here in use of the number of bytes which will be written (Maximum of 256 bytes in a row).

- WriteTimeData1: This value is following the formula:

$$\boldsymbol{WriteTime} = (\boldsymbol{WriteTimeData1} \times \mathbf{2}) / (\boldsymbol{OperatingFrequency})$$

The time the ROM-cells are written is declared in this cell.

Example for frequency of 8 MHz:

Operating frequency	WriteTimeData1(dec.)	Write-Time
8 MHz	200	50 µsec.

- EraseTimeData(1-3): These three values are calculated by the assumption, that in a first try, the erasing of ROM should take 2 seconds of time; if the first erase wasn't successful, the erase-time will be changed to a value about 0.25 seconds and the block will be erased with this time-quanta again, until the whole ROM is erased.

For two seconds erasing, the three values are defined as 0xF5, 0xB, 0x5 at 8 MHz following the formula

$$\mathbf{EraseTime = EraseTimeData1 \times 2^{\frac{(EraseTimeData2 + EraseTimeData3)}{OperatingFrequency}}}$$

The values for TimeData2 and TimeData3 should be left as they are.
Example for Erase-Times of 2 and 0.25 seconds@8 MHz:

OP-Freq.	EraseTimeD1 (dec.)	EraseTimeD2 (dec.)	EraseTimeD3 (dec.)	Erase-Time
8 MHz	245	11	5	2.007 sec.
	244	8	5	0.250 sec.

- ConvTimeData(1-3): The ConversionTimeData defines a value in milliseconds, how long a Flash-cell has to be re-written after an over-Erase has occured.This time the over-erased cells are written-back. A normal time quanta for this matter is about 10ms. This is created by ConvTimeData2 = 0x08, ConvTimeData3 = 0x03 or 0x01 and for 8MHz with ConvTimeData1 = 0x9D following the equation

$$\mathbf{ConversionTime = ConvTimeData1 \times 2^{\frac{ConvTimedata2 + ConvTimeData3}{OperatingFrequency}}}$$

The conversion-Time results in 10 msec.
Example for Conversion-Times of 50 and 10 msec with OP-frequency of 8 MHz:

OP-Freq.	ConvTimeD1 (dec.)	ConvTimeD2 (dec.)	ConvTimeD3 (dec.)	Conv.-Time
8 MHz	195	8	3	50 msec.
	157	8	1	10.05 msec

- **WriteData StoragBuffer StartAddress:** These two bytes define the starting-address, of a RAM buffer. The RAM buffer is used to stocks data for writing into the upper Flash-block and for verifying.

- **RAM-routine StartAddress:** These two bytes contain the address where the user-program is located. It is called while the RTSF-function in ROM-programming is enabled.

- **FlashSwap ReturnAddress:** These cells contain the address, where after the FlashSwap the new start-Vector of the new program-area is located.

- **ReturnInterval TimeData1:** This value adjusts the interval the RTSF-support function is called depending on the external connected frequency. The ReturnIntervalTime is calculated by the formula:

$$\mathbf{ReturnIntervalTime = (ReturnIntervalTimeData1 \times 2)/(OperatingFrequency)}$$

Example for 512 µsec that secures a RTSF-call of less than 1.024 msec.

Return-Interval Time:

OP-Frequency	ReturnIntTime Data1 (dec.)	Return Interval-Time
8 MHz	16	0.512 msec.

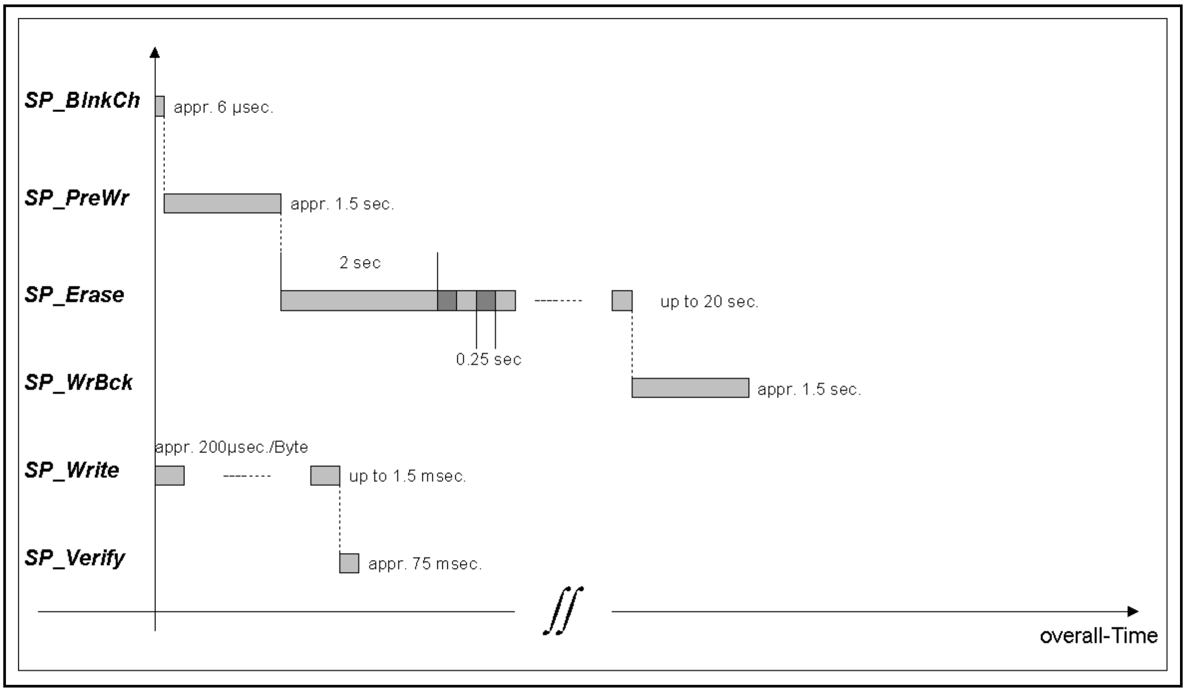
To write this data by user-program, a pointer has to be defined (e.g. pointer_ERAM), which will be anchored to an array located at the pre-defined EntryRAM-address (e.g. array_ERAM).
If the erase-time values from the first try to the repeated try should be changed, following syntax will do this for example:

```
pointer_ERAM = &array_ERAM;
( ... )
pointer_ERAM ->_EraseTimeData1 = RepeEraseTimeData1; // sets the erase-time to shorter
                                                       cycle to prevent
pointer_ERAM ->_EraseTimeData2 = RepeEraseTimeData2; // flash-cells from stressing!
pointer_ERAM ->_EraseTimeData3 = RepeEraseTimeData3;
```

The pointer is linked to the array-structure, then the needed ER-values are set.

Following drawing will show sample values for the single function-times. These values are still preliminary and are intended to be changed. Besides, these values will give intentions for typical times, changes to upper- or lower-bonds will be possible.

Figure 1-7: Sample time-slices for SP-Routines

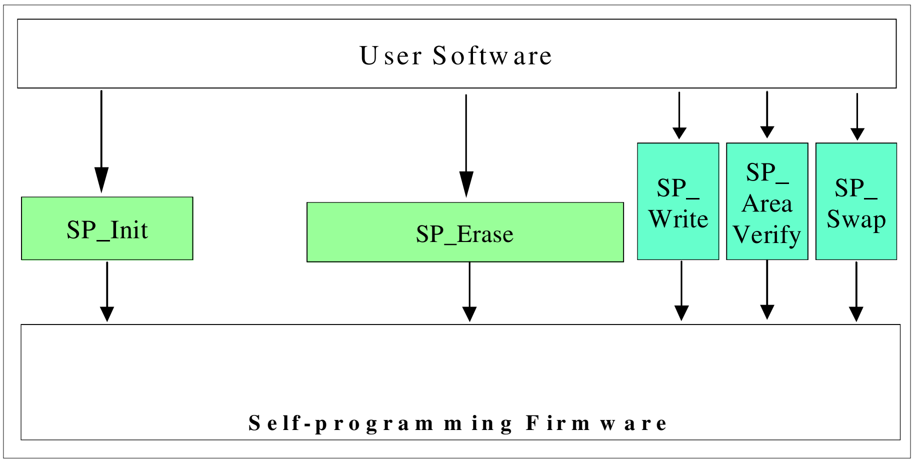


The functions used in the Flash-Self-Programming-library to interface the Self-Programming firmware are shown below.

1.2.6 Functions in the Self-Programming library

This five functions provide all features to do a Secure Self-Programming of the complete Flash-Area.

Figure 1-8: Functions of Self-Programming

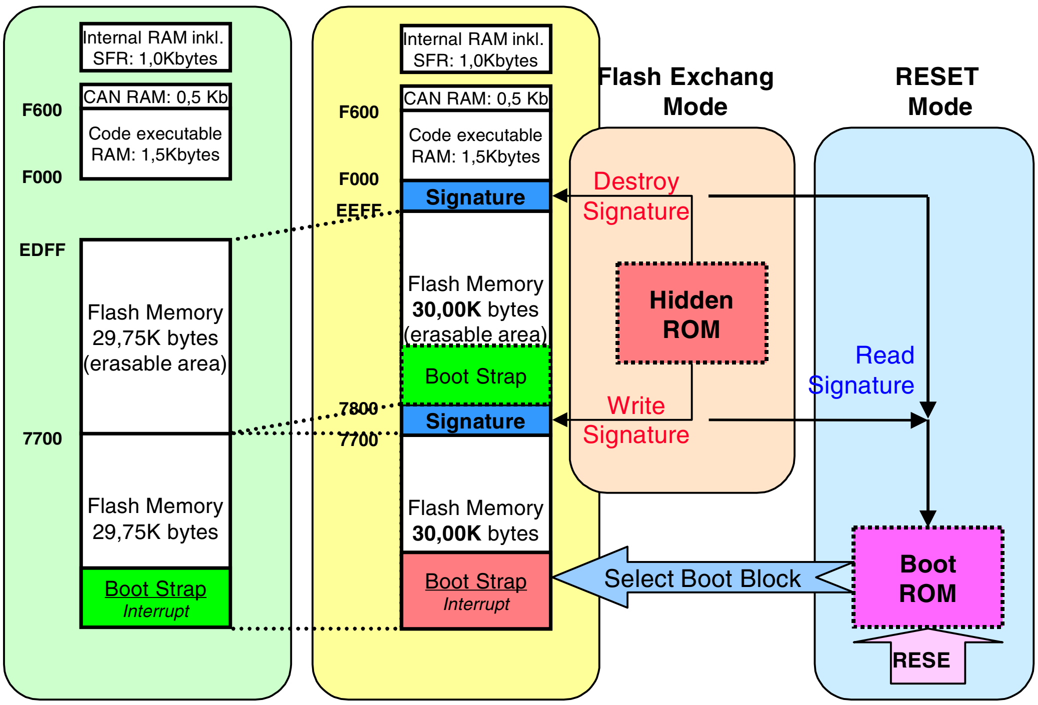


Their functionality will be described later on in this user-manual.

1.2.7 Secure Self-Programming

To guarantee secure Flash Self-Programming, independent from e.g. loss of programming-voltage or loss of device voltage during programming or during switch of the upper- and the lower mem-block, a special function is designed to get at least one memory-block active after RESET.

Figure 1-9: RESET-execution and boot-signature-recognition

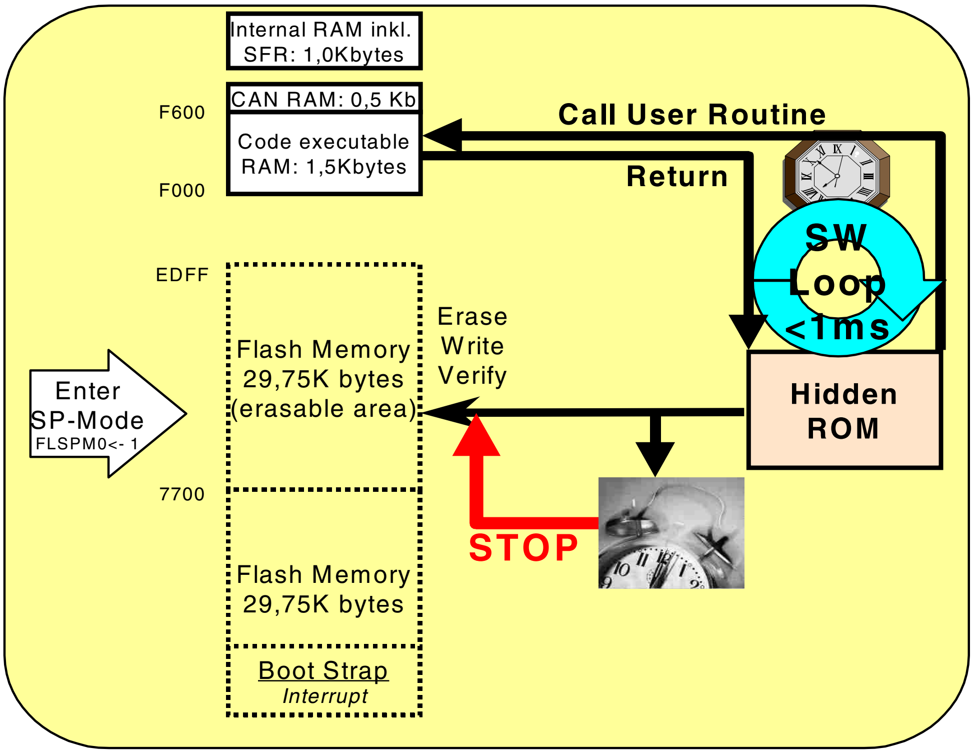


A signature is located on the upper-side of each Flash-block. This signature is written active or non-active, depending on the status the block should have.
The block with the written signature is automatically the lower block. In unfavourable circumstances e.g. voltage drop, the signature of the upper Flash-block might be written and the one of the lower block might not be destroyed.
After RESET, randomly one of the two blocks may be recognized as valid. In this case, the user must prepare his software to recognize whether the original or the new programmed block is containing the actual code

1.2.8 RTSF-Support

The RTSF (so called RealTimeSupportFunction)-support gives the user the possibility to let real time functions like watchdog timer, network management communications or other run while the Self-Programming functions are in use.
To start the RTSF-support by calling the Self-Programming-library-functions, a RTSF-value unequal to 0x0000, equivalent to the start-address of the RTSF-user-function in RAM must be set. This will call the RTSF-function which will set this start-address of the RTSF-user-support-function into the Entry-RAM-Area. For RTSF-usage, a dedicated Area form 0xF000 to max. 0xF5FF in RAM is reserved. The user must define this Area in his.xcl-link-file and has to copy the RTSF-user-function into RAM by himself.
While calling the Self-Programming-routines, in non-equidistant steps lower than 1msec. the support-function is called without halting the Self-Programming-routines.

Figure 1-10: Outline of RealTimeSupportFunction

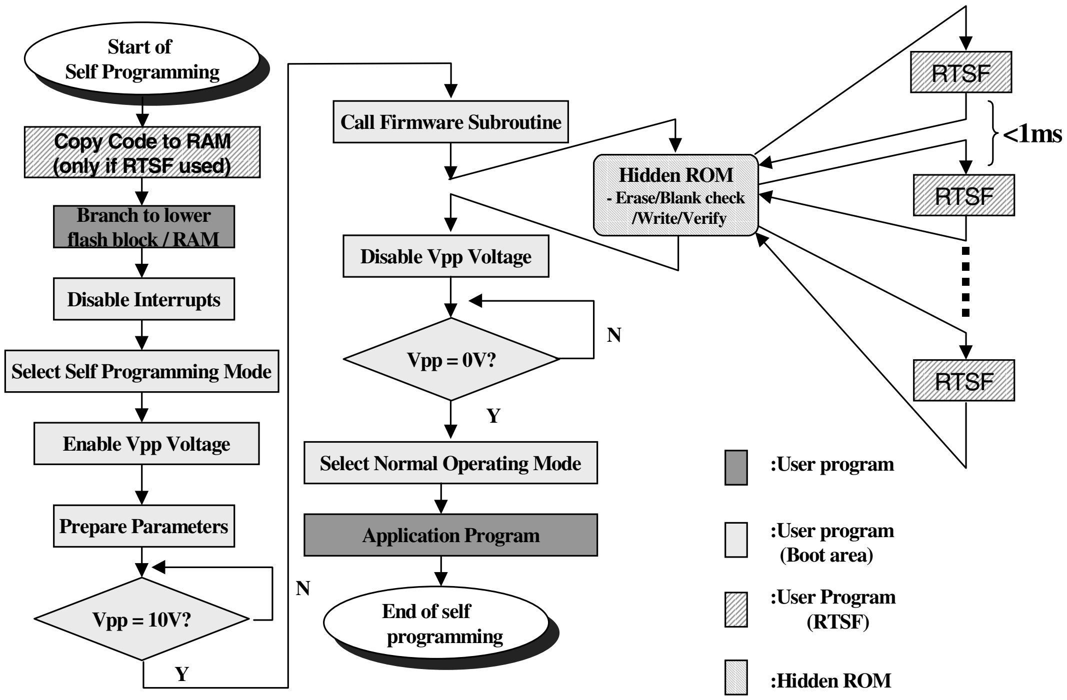


Nevertheless, it is not dangerous to call user-support-functions probably running longer than the execution-interval of the Self-Programming-function. The SP-function is stopped after the execution-interval is ended without any respect to the RTSF-user-function-duration.
Using this feature, no long-durating user-RTSF-support-function can disturb the running SP-procedures.

The user-function resides in the Area from 0xF000 to max. 0xF5FF, equivalent to 1.5 KByte of RAM. This Area must be allocated by the user for RTSF-support.

As a general overview, the whole procedure of Self-Programming is shown in the following diagram; please use this as template for all Self-Programming purposes!

Figure 1-11: Complete Self-Programming procedure



Chapter 2 Software

2.1 Memory-demands

2.1.1 Setup the RAM-mapping

As described before in this document, the RAM-usage has to be canalized for using all features of the Flash-SelfProgramming routines. Some constant memory-layouts are necessary as follows:
An area with a length of 32 Bytes, containing the EntryRAM
The Write Data Storage-Buffer with a defined length of 256 Bytes
The Stack Area, length approximately 40 Byte
The User-Code Area with a max. length of 1.5 KByte, defined from 0xF000 to max. 0xF5FF.

2.1.2 Linker-configuration

In the IAR-workbench the linker-file(xy.xcl) defines the needed areas.
A sample linker-file configuration is shown below:

Figure 2-1: Sample-header-file with mem-configuration

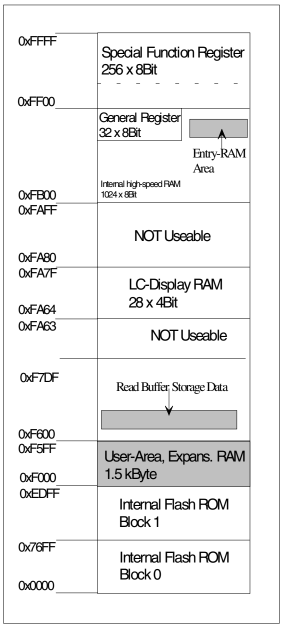
```
//------------------------------------------------------------------------------------------
//                      -LINK.XCL-
//
//    XLINK command file to be used with the 78000 Embedded Workbench
//    using procesor option -v1 memory model option -ms or -mS
//
//------------------------------------------------------------------------------------------
//    Archived: $Revision: 1.1 $
//    (c) Copyright IAR Systems 199
//------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------
// changes for Secure SelfProgramming
// (C) NEC Electronics (Germany) GmbH 2000
//------------------------------------------------------------------------------------------


//------------------------------------------------------------------------------------------
//    Define CPU.
//------------------------------------------------------------------------------------------
-c78000

//------------------------------------------------------------------------------------------
//    Define all user-specific RAM-areas for Secure SelfProgrammin
//------------------------------------------------------------------------------------------

-Z(DATA)ucode=F000-F5FF          // Area to write user-code in RAM
-Z(DATA)MySegER=FE21-FE43        // RAM-segment for Entry-RAM (32byte)
-Z(DATA)MySegRB=F6E0-F7DF        // RAM-Buffer Segment (here 256Byte)
                                 // ! reduces the DCAN t
                                 // ! 12 Receive-Buffers
```

The user-code area is fixed to location F000 tomax. F5FF, the Entry-RAM-Area is defined from FE21 to FE43, resulting in a length of 32 bytes, the Read-Buffer is defined from F6E0 toF7DF, resulting in a length of 256 Byte.

2.1.3 Resulting memory-layout

By allocating the needed segments above, the following Self-Programming memory-layout is generated:

Figure 2-2: Memory-outline using SP-settings



2.2 Prepare Self-Programming

This chapter will describe the conditions to get a full running system with Self-Programming.

2.2.1 Pre-conditions

To get the Software running, the following should be defined before starting programming.

Is there need of a User-function, that provides the ability of Real-Time Support?
If this function is needed, define the approximate length of the program that will reside in the memory-Area from 0xF000 to 0xF5FF.

Which protocol should responsible to get the data to program from outside into the chip?
Define the protocol, using e.g. UART, CSI, CAN.

Where should the incoming data be stored?
Make sure, that there is at least one RAM-area defined with a maximum length of 256 bytes, where the incoming data can be stored, until it will be written by library-calls into the Flash, so that new data can be acquired.

Where should the Entry-RAM-Area reside?
Make sure that an Area in RAM with a length of 32 bytes is reserved, so the data necessary for the Self-Programming-mechanism can be stored.

2.3 Application Programming Interface Functions

2.3.1 Standardized Input- and Output-parameters

All functions will use very similar In- and Out-Parameters, which will be described as follows.

Input-Parameters

- RTSF-Parameter

The input-parameter used in every function is the _RTSF-variable.

If this variable is set to 0x0000 while calling the depending SP-library-function, no user-RTSF-function will be called while making the SelfProgramming-function.

Otherwise, if there is need of the RTSF-Function, the starting-address of this user-function has to be defined in the function-call; usually, this will define address in the range from 0xF000 to max. 0xF5FF

- SPWriteStartAddress

This variable is used by the Write-function and defines the target-address in the upper Flash-ROM-block, where writing data from the RAM-buffer is started; any value from 0x7800 onwards to 0xEE00 minus SPNoByte is valid.

- SPNoByte

For the Write-Function, this value defines the number of bytes to be written into the Flash-Memory starting from SPWriteStartAdr onwards. Valid values are from 1 to 256.

- SPBuffStartAddress

Defines the Start-address of the RAM-buffer. This RAM-buffer contains the data for the write-routine and is used by the Area_Verify-routine, too.

- SwapRetAdr

This address defines the starting-address of the new program, which will be called after the upper and the lower block were swapped; please remember that this must not be the address the microcontroller jumps to after executing the RESET-procedure.
While the Swap is executed, the signature of the upper block is written and the signature of the lower block is destroyed, so that after a RESET-Function is released, the old upper-block retains his new lower-position and the new program is executed by taking the new start-vector at address 0000.

Output-Parameter

Error-Flag
All functions will return the same parameter, but with different values. The different values are worked within the SP-library, but errors, which are not re-workable will be directed to the calling program. The error-codes are listed in the following:

Figure 2-3: Return-Error-Messages

```
//--- Error-Messages
#define ParamError      0x80
#define VerifyError     0x10
#define WriteError      0x08
#define OverEraseError  0x04
#define BlankCheckError 0x02
#define NoError         0x00
```

Following, a short description of the error-messages follows:

- Parameter-Error: This error is defined, if some of the values, defined in the Entry-RAM-Area, are not compatible to the function-execution, e.g. Block-No != 1 or write 20 Bytes to 0xEDF0.
- Verify-Error: This error is defined when internal verification of written data is not executed correctly.
- Write-Error: If the data from RAM to Flash-ROM cannot be written correctly, this error will be shown.
- OverErase-Error: This error is returned if any cell is overerased. This could happen in following cases:
  - erasing takes too long
  - erased cells are erased again
  - cells, programmed with FF, (equals to never programmed), are erased.

The last two cases can be avoided by using the prewrite-function before each erase-function call. Depending of the over-erase status of the cell, an over-erase error could be corrected by using the WriteBack-function.

- BlankCheck-Error: If the internal Blank-Check finds non-blank memory-cells, this error-code is generated.
- No error: No error is encountered.

2.3.2 SP_Init

Figure 2-4: Header of Function SP_Init

```
extern char SP_Init(unsigned int _RTSF_Adr);
//============================================================================
//    Purpose: This function initializes all ENTRY-RAM with global data by calling
//             library-functions with the pre-defined time-values out of this
//             header-file.
//             SP_Init will clear the Entry-RAM-Area except (HL + 0x0e)..(HL + 0x13),
//             and copies the user-defined time-data in Entry-RAM.
//             RTSF during the function-call can be enabled!
//             This function allows the user to place own user-functions during ROM-
//             write-process (e.g. watchdog); interrupts can be polled only!
//============================================================================
```

Like described in the above picture, the In- and Out-Parameters are as follows:

Input-parameters:	_RTSF_Adr: 0x0000 when no RTSF-support is needed
output-parameters:	RB3_B: error-code stored in register B of Bank 3

The function calls several firmware-routines to initialise and check the EntryRAM. If a RealTimeSupportFunction is requested by the input-parameter, the RTSF-function is automatically called within the used firmware-routines.
Default values out of the Firmware-ROM, that are based on an 8MHz clock are written into the Entry-RAM-Area. These are 50 µsec, 2 sec and 10 msec for the three parameters write-, erase- and conversion-time.
After setting these values and testing if the settings are in a defined range, a return-value is created and the function is ended.
If the values written by the SP_Init-function doesn't fit into users need, they must be rewrittenmanually. With the given start-address and the offset from chapter “The Entry-RAM” o n page7, all recent values can be changed.
The Start-Address of Entry-RAM must be defined in the.xcl-file, because all library-functions use this address to get data out of the Entry-RAM and start the Flash-Programming!

The following table shows an example for an Init-routine.

```
er->_ByteInFlash_BlockNo     = BlockNo;              // init BlNo / No of Byte in Flash
er->_WriteTimeData1          = WriteTimeData1;       // init Time to write Data
er->_EraseTimeData1          = FirstEraseTimeData1;  // init Erase Time Data in three
er->_EraseTimeData2          = FirstEraseTimeData2;  // vars
er->_EraseTimeData3          = FirstEraseTimeData3;
er->_ConvTimeData1           = FirstConvTimeData1;   // init conversion Time data in
er->_ConvTimeData2           = FirstConvTimeData2;   // three vars
er->_ConvTimeData3           = FirstConvTimeData3;
```

This can be done, if the standardized values, written into the Entry-RAM by the library-Initilization, don't fit into the users special needs.

2.3.3 SP_Erase

Figure 2-5: Header-file of Erase-function

```
extern unsigned char SP_Erase(unsigned int _RTSF_Adr);
//==========================================================================
//    Purpose: First, this function blank-checks block no. zero if it is blank (FFh).
//             If successfull, function is aborted with parameter OK. Returns Blank-
//             Check-Error if any cell is not blank.
//             If blankcheck fails, SP_prewrite is started which writes all
//             flash-bytes of the upper block with (00h) to avoid erasing of erased
//             cells (overerase).There is no check for write-access.
//             Afterwards FirstErase TimeData is copied to the entry-RAM and
//             SP_AreaErase is started. With this function the routine starts
//             the erase of the whole upper flash-block, including the signature
//             for the specified time. It's possible that more streams are needed
//             for the whole flash-block to erase. For this case, all function-calls
//             afterwards are repeated with RepEraseTimeData.
//             By reaching MaxEraseTime, SP_Erase is finished with return-value
//             BlankCheckError.
//             In case of OverErase, function SP_AreaWriteBack is called to try
//             repair of over-erased cells. For the first time, the value FirstConvTime
//             is used. If erasing with this time-quanta wasn't successful, a value
//             for repeated erasing is used instead.
//             The erasing will be ended either by succefully blankcheck or reaching
//             the maximum erase- or conversion-time and returning the related error-code.
//==========================================================================
```

The Erase-function is the most complex function of the library-set. The only input-parameter needed is the RTSF-usage-value.

Input-parameters:	_RTSF_Adr: 0x0000 when no RTSF-support is needed
Output-parameters:	RB3_B: error-code stored in register B of Bank 3

The output-parameter will give information about the status of Flash-block-Erasing. Between the calling and the ending of the erase-function, several options are calculated in the Erase-function itself. A short overview should give the user an introduction what the Erase-function will do internal:
First of all, a blank-Check is done to determine, whether the ROM-block to be flashed is already empty or not.
If the Area is blank, the erase-function is finished with no-error. If there are non-blanked cells, the Pre-Writing of the area is initiated. This Pre-Writing should preserve the ROM-Area from OverErase-Errors by writing 0x00 to all cells.
After this Pre-Write is done, the full upper ROM-Area will be erased. To get a fast time for erasing and a

Type of Erasing	internal SP-lib. Variable-Name	Time-quanta
Initial Erasing	FirstEraseTimeData(1..3)	apr. 2 sec
Repeated Erasing	RepeEraseTimeData(1..3)	apr. 0.25 sec

minimum stress for the flash-block the erasing is done in an iterative way. Between each step, an erase-success is checked by a blank-check.

Type of Erasing	internal SP-lib. Variable-Name	Time-quanta
Initial Erasing	FirstEraseTimeData(1..3)	apr. 2 sec
Repeated Erasing	RepeEraseTimeData(1..3)	apr. 0.25 sec
Max Erasing	-	30 sec

While the first Erase-cycle is not successful, the writing-parameters are changed to the shorter repetitive values to start the next erase-iteration.

In case of OverErasing, the SP_Erase-subroutine will write-Back the FlashROM. This means, that the write-back-function tries to repair the over-erased cell by kind of special writing.
The SP_Erase-function will be ended either by successful Blank-Check or reaching the maximum erase- or conversion-time. By reaching the maximum times an error-code is returned.

2.3.4 SP_Write

Figure 2-6: API-Header for SP_Write

```
unsigned char SP_Write(unsigned int SPWriteStartAdr, unsigned char SPNoByte,
unsigned int SPBuffStartAdr, unsigned int _RTSF_Adr)
//===========================================================================
//    Purpose: Calls the firmware-routine ByteWrite thats writing up to 256 bytes
//             to the flash out of the RAM-buffer.
//             Writing will be re-tried up to ten times before ending with write-
//             error.
//             RTSF can be enabled.
//===========================================================================
```

As described in the above header-file, this function uses more than just the RTSF-input-parameters:

Input-parameters:	output-parameters:
SP_WriteStartAddress: Sets address where in ROM data-writing should be started	**RB3_B:** error-code stored in register B of Bank 3 **Write-error:** Writing to at least one bit was not successful. **Parameter Error:** WriteStartAdr + SPNoByte exceeds write-able Flash-Area
SPNoByte: Defines the number of Bytes to write in ROM	
SPBuffStartAdr: Declares the start-address where data in RAM resides	
_RTSF_Adr: 0x0000 when no RTSF-support is needed	

The SP_Write-function fetches data from pre-defined RAM-area and writes it into the upper Flash-Block at starting address, defined while the function is called.
If the RTSF-support is requested, the start-address of the RTSF-support-function is written to the according Entry-RAM-Bytes.
With the input-parameters, the EntryRAM-values Flash Memory Start-Address, No. of bytes written in flash-memory and WriteData_StoragBuffer_StartAddress are declared.

The write-Time, used for the write-in of data, is calculated as described in the following formula:

$$\mathbf{WriteTime} = (\mathbf{WriteTimeData1} \times \mathbf{2}) / (\mathbf{OperatingFrequency})$$

With an operating-Frequency of 8 MHz and a Data1 of 200, the basic WriteTime will amount to 50 µsec.

2.3.5 SP_AreaVerify

Figure 2-7: API-header for the SP_AreaVerify-Function

```
extern unsigned char SP_AreaVerify(unsigned int SPBuffStartAdr, unsigned int _RTSF_Adr);
//===========================================================================
//    Purpose: This function uses firmware-routine AreaVerify to measure if the data
//             in Flash is able to stand a 10-year retention.
//             This is done in steps of 256 Bytes.
//             Attention: This is done with use of the 256 Bytes of the defined RAM-
//             buffer.
//             Please take care, that writing of data and Verifying or writing different
//             data with verifying doesn't go apart too much.
//             RTSF can be enabled.
//===========================================================================
```

The Verify-function needs also more than one calling-parameter. The ROM-address, where the temp-data will be written to, is still defined.

The parameters are listed in the following table:

Input-parameters:	output-parameters:
SP_BuffStartAdr: Sets address of RAM-buffer where temporarily data is stored	**RB3_B:** error-code stored in register B of Bank 3
_RTSF_Adr: 0x0000 when no RTSF-support is needed	

The verification of the whole Flash-Area is done by this function. The input-parameter SP_BuffStartAdr is copied into EntryRAM-Area to the WriteDataStorageStartAdr.
The firmware-routine does a byte-to-byte-compare of the whole Flash-block in 256-Byte-steps. Therefore, the RAM-buffer is used to copy the data out of the Flash into the RAM and do the comparison. If the check wasn't successful, an error is generated and the test is ended.
Please take care, that the writing of the data and the AreaVerify shouldn't go apart to long. A successful test will guarantee a data-security of 10 years.

2.3.6 SP_Swap

Figure 2-8: Header-file of SP_Swap-function

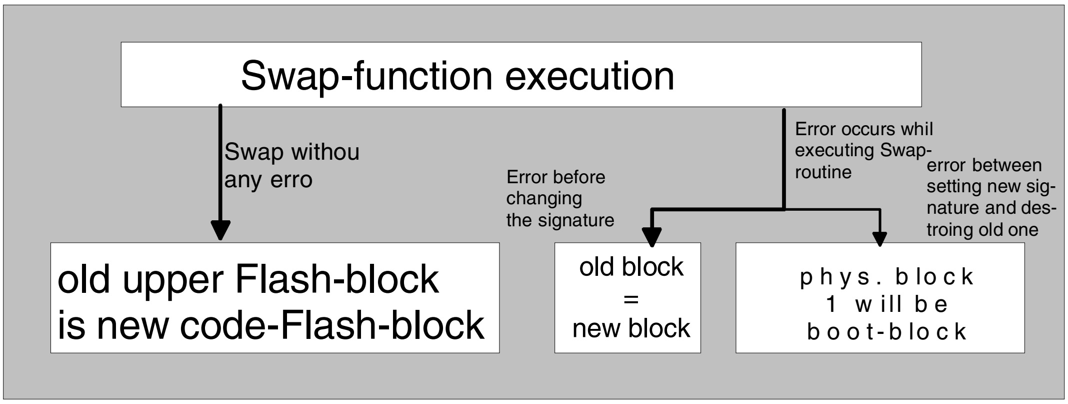
```
extern unsigned char SP_Swap(unsigned int SwapRetAdr, unsigned int _RTSF_Adr);
//=============================================================================
//    Purpose: Calls the firmware swap-routine that moves upper flash-block with
//             the lower flash-block. After this remapping, the customer-code at
//             SwapRetAdr is started.
//             RTSF can be disabled.
//=============================================================================
```

The Swap-function should be called after all needed data is written from RAM to Flash, so a full-featured program is located in the upper memory-block. This following, the upper- and the lower block must be switched, so the former lower-block can be erased and written to access the whole Flash-ROM-Area.

Input-parameters:	output-parameters:
SwapRetAdr: Sets address where program counter after swap has to point to	**RB3_B:** error-code stored in register B of Bank 3
_RTSF_Adr: 0x0000 when no RTSF-support is needed	

If the Swap is executed, the program counter is set to the SwapReturnAddress at leaving the firmware-routine. The starting-address of the related program should be located here.
In case of error while executing the Swap-function (e.g. losing the VPP), following figure declares, which ROM-block will actual be the upper ROM-block:

Figure 2-9: Different effects of Swap-routine



[MEMO]

Appendix A Example-programming-sequence

To guide the user by programming the Flash-Device with the on-chip Secure Self-Programming, a sample programming-sequence is shown.
First of all, the linker-file should be configured to specify the used RAM and Flash-Areas.

Figure A-1: Example for Linker-file in IAR-workbench

```
//-------------------------------------------------------------------------------------------
//                              -LINK.XCL
//
//     XLINK command file to be used with the 78000 Embedded Workbenc
//     using procesor option -v1 memory model option -ms or -mS.
//
//-------------------------------------------------------------------------------------------
//     Archived: $Revision: 1.1 $
//     (c) Copyright IAR Systems 1997
//-------------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------------
// changes for Secure SelfProgramming
// (C) NEC Electronics (Germany) GmbH 2000
//-------------------------------------------------------------------------------------------


//-------------------------------------------------------------------------------------------
//     Define CPU.
//-------------------------------------------------------------------------------------------
-c78000

//-------------------------------------------------------------------------------------------
//     Define all user-specific RAM-areas for Secure SelfProgramming
//-------------------------------------------------------------------------------------------

-Z(DATA)ucode=F000-F5FF              // Area to write user-code in RAM
-Z(DATA)MySegER=FE21-FE43           // RAM-segment for Entry-RAM (32byte)
-Z(DATA)MySegRB=F6E0-F7DF           // RAM-Buffer Segment (here 256Byte)


//-------------------------------------------------------------------------------------------
//      Define all other Self-Programming related stuff
//-------------------------------------------------------------------------------------------

-Z(CODE)USERCODE=4000-45FF          // Area where user-code in ROM resides
-Z(CODE)SWAPCODE=4600-4BFF          // Area where Code for Swap resides
```

The used linker-file defines following Areas:

- **Ucode:** Area, where the User-code used for the RTSF-support-function will reside. Not only in this example, this Area should be from 0xF000 to max. 0xF5FF.
- **MySegER:** Area which includes the Entry-RAM-Area; the example defines this from 0xFE21 to 0xFE43
- **MySegRB:** This Area resides in RAM and stores the data read by a protocol like CAN. The example sets this value from 0xF6E0 to 0xF7DF.
- **USERCODE:** By first programming, this area contains the data that will be copied as RTSF-user-support function into the Ucode-Area.
- **SWAPCODE:** This area contains code which can be copied to the Flash and be called after both Flash-blocks have been swapped.

To complete the example, the non-library-version of the MAIN.c-file is given; here certain values are defined for the CAN-ASSP3, which should be carried by the user to guarantee Self-Programming-functionality:

Figure A-2: Example for Self-Programming MAIN.c-file

```
/*===============================================================================*/
/* PROJECT   = Self-Programming Library for CANASSP3  NEC Electronics (Germany)GmbH*/
/* MODULE    = MAIN.c                                              Kanzlerstraße 2*/
/* VERSION   = 1.0                                             D-40472 Duesseldorf*/
/* DATE      = 30.06.2000                            R.Laschewski,NEC-EG/APD.AMAC/CSD*/
/*===============================================================================*/
/*                              C O P Y R I G H T                                */
/*===============================================================================*/
/* Copyright (c) 2000 by NEC Electronics (Germany) GmbH.   All rights reserved.   */
/*===============================================================================*/
/*Purpose:    This is the MAIN-file for testing the functions given in SP-library */
/*            All functions that provide Firmware-functionality are called with   */
/*            propper parameters, and an example is tested to copy user code from */
/*            set ROM-area to RTSF-depending RAM-area!                            */
/*                                                                                */
/*===============================================================================*/
/* Enviroment:   Devices:        Test: later:    CANASSP3                         */
/*               Assembler:      A78000          Version 1.13A                    */
/*               C-Compiler:     ICC78000        Version 3.13A                    */
/*               Linker:         XLINK           Version 4.50                     */
/*               Debugger:       ID78k0NEC       Version 1.20b                    */
/*===============================================================================*/
/*Changes:                                                                        */
/*===============================================================================*/

#include "SP_lib.h"
#include "SP_API.h"
#include "io78082x.h"                  // Include SFR definitions
#include "SP_exam.h"

void main(void)
{
  PCC = 0x00;                            // sets system-frequency
  IMS = 0xCF;                            // sets system RAM and -ROM
  IXS = 0x08;                            // sets system expansion-RAM
  SP_Init(0xF400);                       // parameter: RTSF-Start?!
  SP_Erase(0xF400);                      // 0x0000 for testing-purposes!, parameter: RTSF-Start?!
  SP_Write(0xEDFF, 128, WriteDataStorageStartAdr, 0xF400);// parameter: where to write, length,.., RTSF
  SP_AreaVerify(WriteDataStorageStartAdr, 0xF400); // parameter: .., RTSF
  SP_Swap(0x7700, 0xF400);               // adress to begin after swap, RTSF
  SP_Init(0xF400);
  SP_Erase(0xF400);
  SP_Write(0xEDFF, 128, WriteDataStorageStartAdr, 0xF400);
  SP_AreaVerify(WriteDataStorageStartAdr, 0xF400); // parameter: .., RTSF
}
```

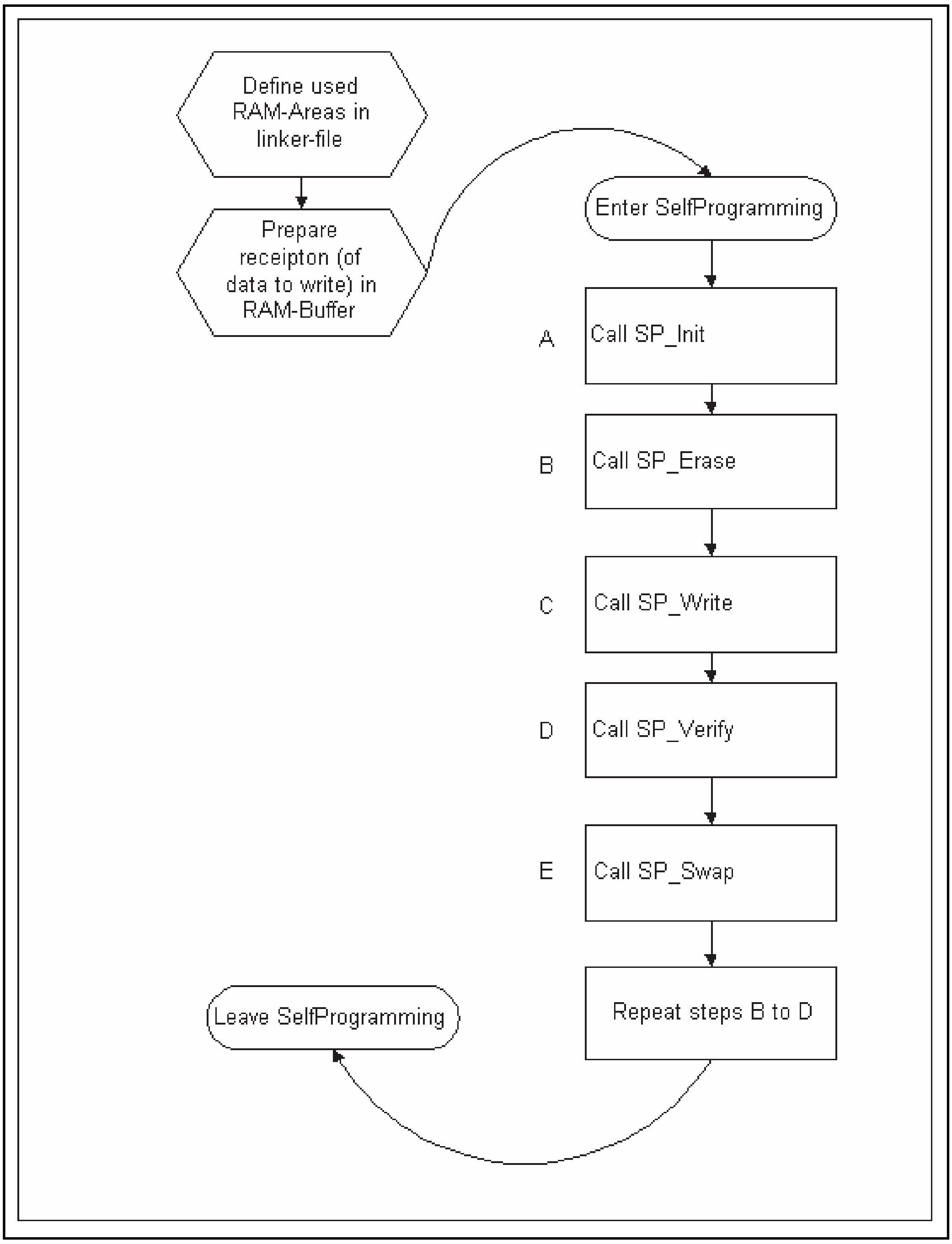
The SP_lib.h file is part of the library, the SP_API.h-file is delivered with the library; herein are defined values which can be changed to fit into users needs.
The used registers should be set in a way like shown above to provide the user with the needed RAM, Flash and system frequency.
Following this, the Self-Programming functions are called in the described way. Please note, that there are only 128 bytes copied for this example; that wouldn't be enough to support further Self-Programming-functions!

In short terms, the following sequence must be applied:

- Define used system-RAM- and -Flash-Areas
- Use SP_Init to initialize the Entry-RAM-Area with valid values
- Use SP_Erase to erase the upper Flash-Memory-block
- Be sure that the data to be programmed in the formerly blanked Flash-block is located in the RAM-Buffer
- Call SP_Write to write the data from RAM to Flash
- Get the Verification of the written data with SP_Verify
- Do a Swap using SP_Swap to activate the new written program
- Program the now new upper-block with data from RAM-Buffer with steps described above.

As description, the following flowchart gives an overview how to program the Flash-ROM:

Figure A-3:   Sample-Flowchart for complete SP-description

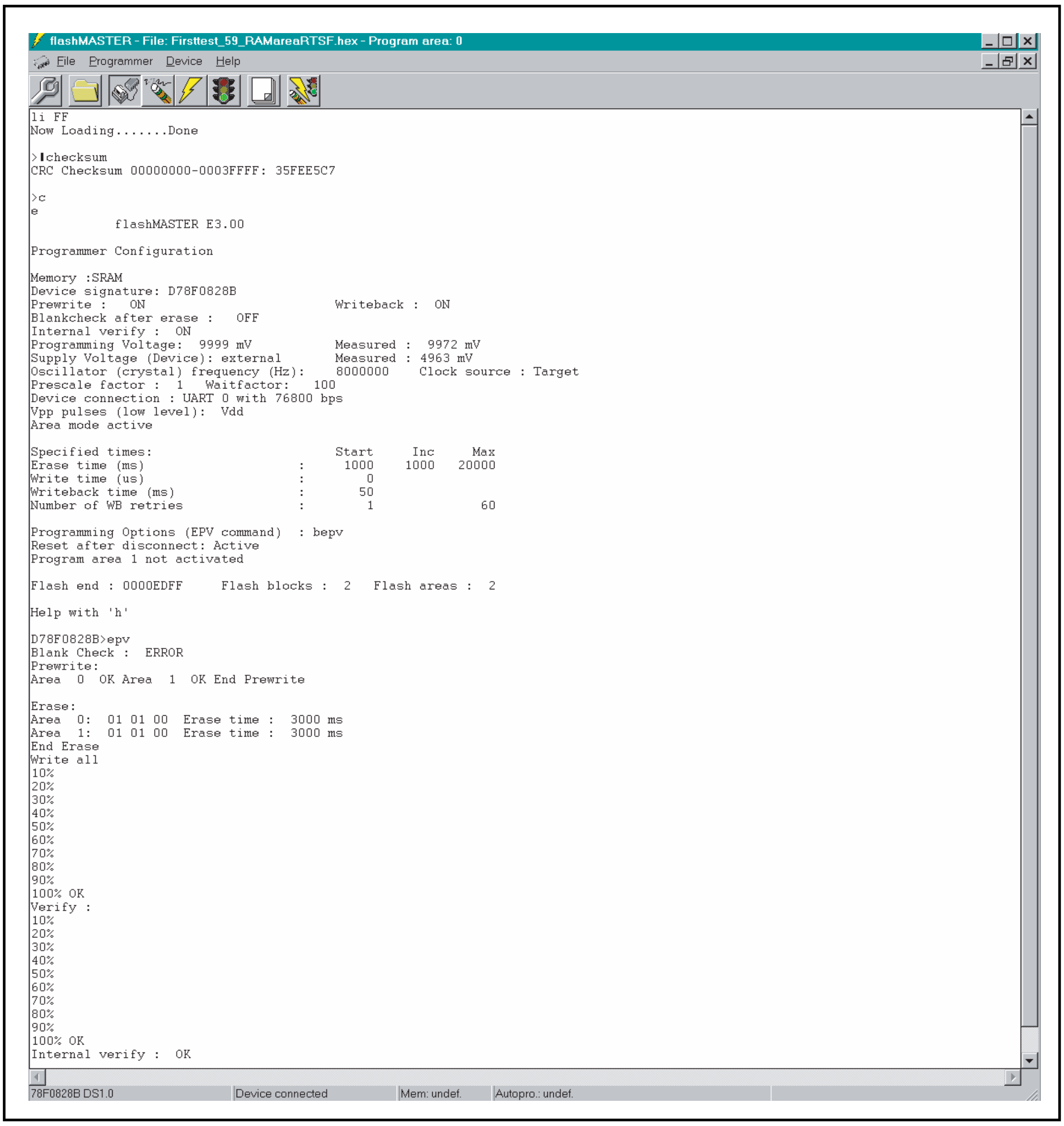


After the declaration is done, the user-program should define a protocol which can carry the data into the MySegRB-Area in blocks of 256 Byte.
When the area is filled, the Self-Programming mode has to be entered by calling the belonging SP-Routines.
First, the steps A to E in the above given Flowchart are executed to get a fully programmed upper Flash-memory-block. With swapping both blocks, the new upper-block can be programmed.
Please remember, that a boot loader, containing the library for Self-Programming, has to be included in the new-programmed application for future Self-Programming.

Appendix B Programming with FLASH-MASTER

The following sequence shows the monitor of the flash-master, connected to a PCvia any Terminal-program:

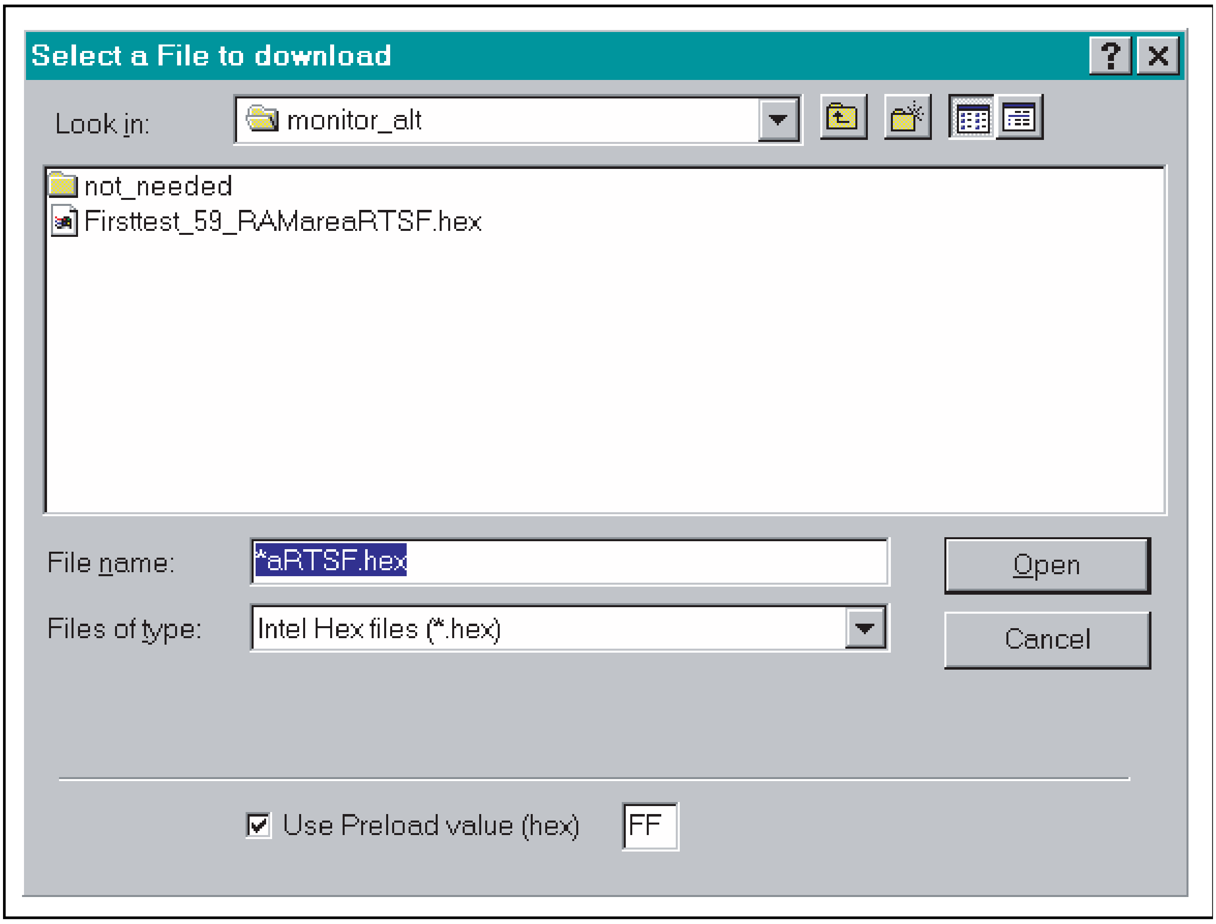
Figure B-1: Output-sequence of Flash-Master-Software



To run the above shown Flash-Master-sequence, a hex-formatted-file must be generated.

This hexfile is downloaded into the Flash-Master.

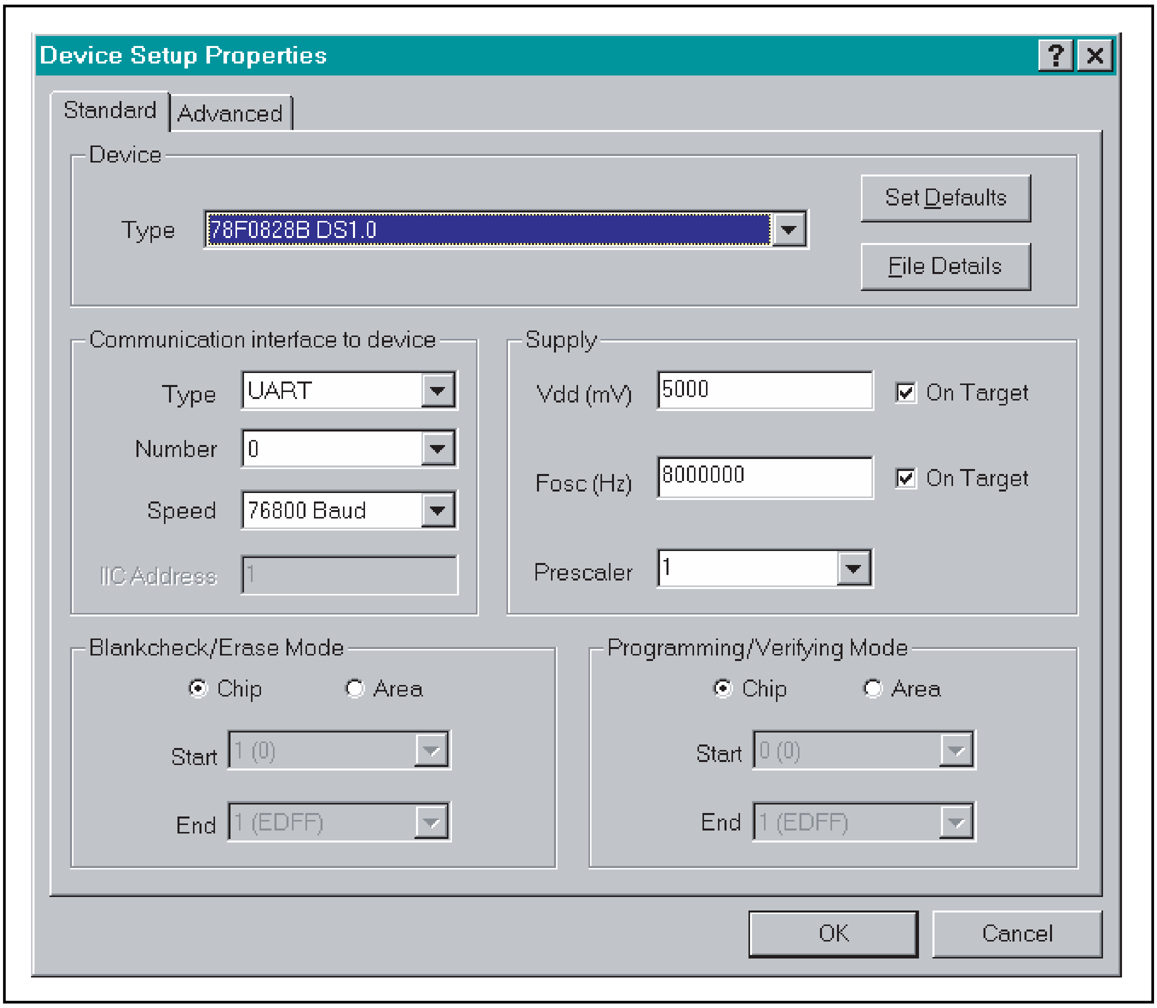
Figure B-2: Download of intel-hex-formatted file into the Flash-Master



Than the combined sequence BlankCheck - PreWrite - Erase - (WriteBack) is executed.
After this is finished, a system-RESET has to be done to activate the formerly written program.

The settings for the device are shown in the last picture:

Figure B-3: Device-Properties



Using this sequence, the device can be programmed with the Flash-Master as easy as with the On-Chip Secure Self-Programming-functions.

[MEMO]

Appendix C Revision History

- Documentation

Version	Description
V 1.0	Initial Documentation

- Self-Programming Library

Version	Description
V 1.0	Initial Library-release

[MEMO]

Appendix D Related Software

To use the described SelfProgramming-features, the following Software-parts are necessary:

Description	Name
Header-file with API-Information	SP_API.h
library-file with described library-functions to use with 78F0828b	SP_lib.r26
Emulation-library	emulator.r26
Device-file for 78F0828b	Df0828b.h
Sample Link-file for use with the IAR-workbench	SP_Lnk.xc
library with outsourced standard-library-calls	SP_l06fit

[MEMO]

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Technical Documentation Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-6465-6829

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: 02-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: 044-548-7900

I would like to report the following error/make the following suggestion:

Document title:

Document number: Page number:

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐
Organization	☐	☐	☐	☐

CS 99.1