NEC

Application Note

78K0/Kx1+ Self-Programming

Document No. U18053EU1V0AN00

The information in this document is current as of April 2006. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC sales representative for availability and additional information.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such NEC Electronics products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.

Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of customer's equipment shall be done under the full responsibility of customer. NEC Electronics no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.

While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.

NEC Electronics products are classified into the following three quality grades: “Standard”, “Special” and “Specific”.

The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated “quality assurance program” for a specific application. The recommended applications of NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.

"Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.

"Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).

"Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is “Standard” unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact NEC Electronics sales representative in advance to determine NEC Electronics 's willingness to support a given application.

Notes:

1. "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
2. "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02.10

Revision History

Date	Revision	Section	Description
04-2006	—	—	First release

Contents

1. Introduction

The 78K0/KX1+ microcontrollers (MCUs) have the ability to program their own flash memory. This function, flash self-programming, is based a set of C language-based functions that hide a lot of detail in the flash self-programming process. These C functions are described in this manual and used to illustrate a typical flash self-programming procedure for the µPD78F0148H MCU.

This manual does not deal with obtaining the data or code to be flash self-programmed into the MCU. For that information, see the *78K0/KX1+ Bootloader Application Note* (document number U18054E). For additional information about the flash self-programming process that underlies the provided C functions, refer to the *78K0/Kx1+ Flash Memory Self-Programming User's Manual* (document number U16701E).

2. Operating Modes

The MCU has three operating modes: normal, A1, and A2.

2.1 Normal Operation

In normal operation, the MCU executes a user program and cannot self-program itself.

2.2 A1 Mode

A1 mode provides access to hidden ROM functions for flash self-programming. With the FLMD0 pin set high, you can call the hidden ROM functions for erasing, writing, and so forth using these MCU resources:

- Timer 50
- Register bank 3
- Entry RAM (48 bytes)
- Data buffer (4–256 bytes)
- Stack (32 bytes)

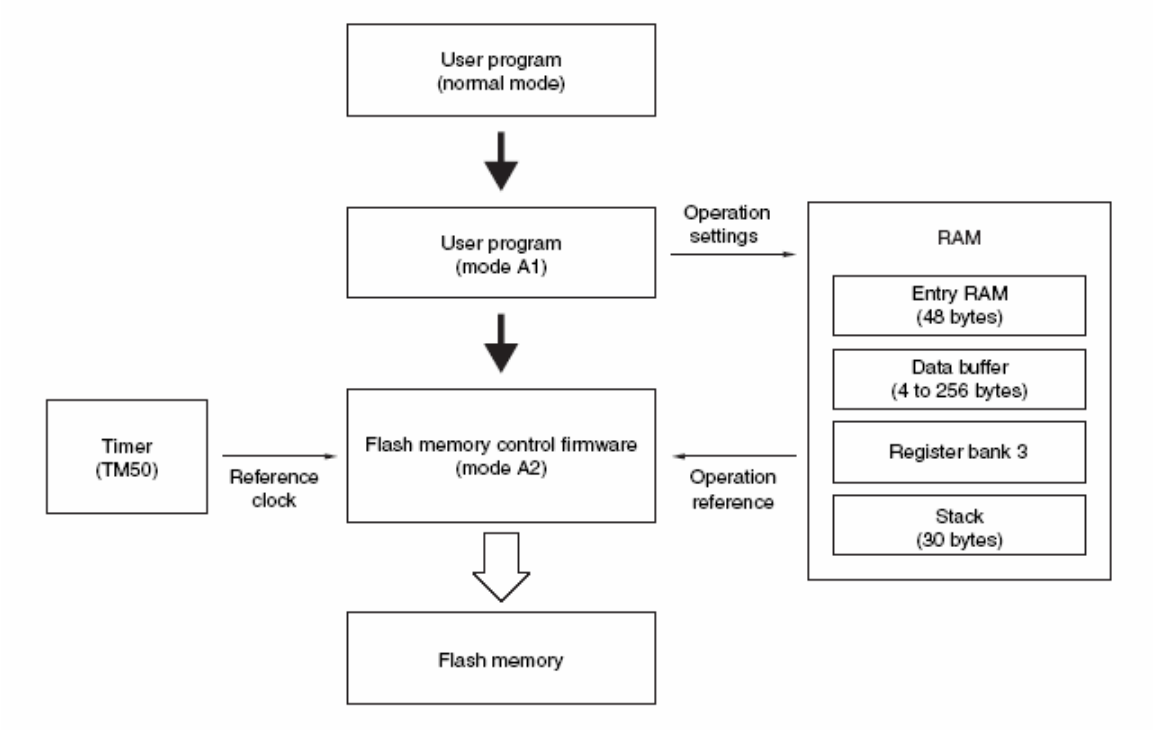
All of the flash memory can be blank checked, erased, written to, and so forth, but the executing code that calls the hidden ROM functions must be located between addresses 0080H and 7FFFH because the hidden ROM functions use addresses starting with 8000H upward. Interrupts should be disabled in A1 mode.

When finished with the hidden ROM functions, immediately return to normal operation where the hidden ROM functions are inaccessible.

2.3 A2 Mode

When the MCU is executing the hidden ROM functions, it is in A2 mode. When the MCU is finished executing the A2 mode function, it returns to the A1 mode calling routine.

Figure 1. Self-Programming Modes



3. Flash Microcontroller Architecture

3.1 Flash Blocks

The µPD78F0148H MCU's flash memory is divided into blocks of two kilobytes (KB). as shown in Figure 2. This is the smallest amount of memory that can be blank checked, erased, or verified. The available 60 KB of flash memory between addresses 0000H and EFFFH are divided into thirty 2 KB blocks (block 0 to block 29).

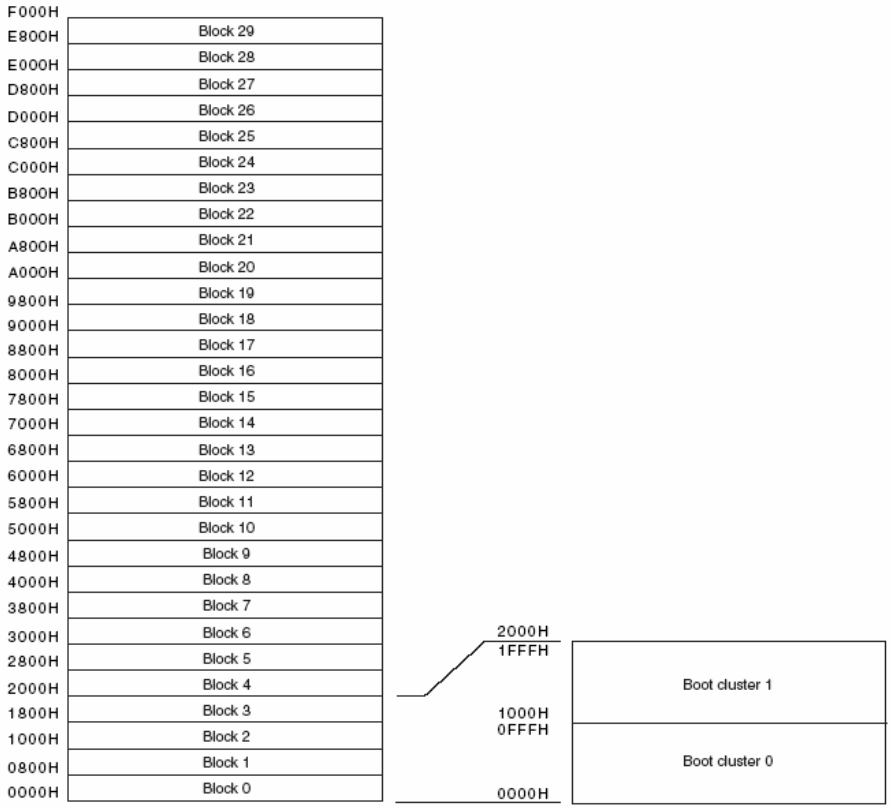
3.2 Boot Clusters

The first four blocks of flash memory are divided into two 4 KB boot clusters, called boot cluster 0 and boot cluster 1, that are used along with the bootswapping feature to allow a bootloader to download a new bootloader that overwrites the old one. For more details on this, see the *Bootloader Application Note*.

3.2 Flash Words

One four-byte word is the smallest amount of memory that can be written in the data buffer, which holds a maximum of 64 words (256 bytes).

Figure 2. Flash Memory Structure of µPD78F0148H



4. Hardware Requirements

The flash self-programming function uses a single-voltage process, and only the MCU's V_{CC} power is required. The two FLMD0 and FLMD1 pins must be managed as part of the flash self-programming process.

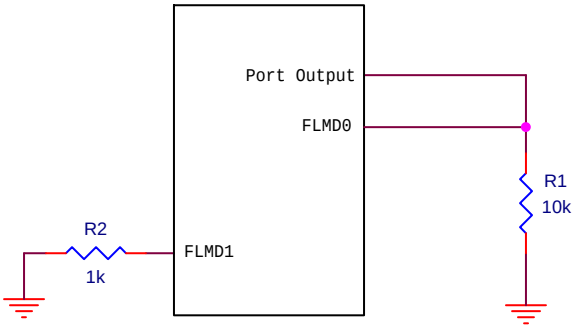
4.1 FLMD0 Pin

Pin FLMD0 normally is low but must be pulled high for flash self-programming. The recommended configuration is shown in Figure 3, where FLMD0 is tied to a port pin configured as an output and then to ground via a resistor. This way the FLMD0 pin can be set high or low by setting or clearing the output port pin.

4.2 FLMD1 Pin

Pin FLMD1 should be kept low. If FLMD0, FLMD1 and RESET are all high, the MCU enters a test mode. As FLMD0 is made high for flash self-programming, it is advisable to keep theFLMD1 pin low. Although this pin can alternate as a port pin, it is probably advisable to tie it to ground via a resistor and leave the port configured to the input state.

Figure 3. FLMD0 and FLMD1 Connection



5. Software Functions

A set of C functions has been developed to make it easier and more intuitive to perform flash self-programming. To do so, add the **sfp.c** source file to your application and include the **sfp.h** header file in your application source files.

Example:

```
#include "sfp.h"      // include self-flash programming routines
```

As the self-flash programming routines use in-line assembly language to interface to the hidden ROM functions in A2 mode, you must specify a C compiler option for the **sfp.c** source file.

1. Right-click the file name in PM Plus and selecting **Special Compiler Options** to display the C compiler options.
2. Select the **Output** tab and click the **Create Assembler Source Module File** option.
3. Click **Apply** and then **OK**.
4. If you have done this successfully, the icon for the **sfp.c** source files should now change to a green color with “OPT” (for optimized) written across it. Note that when you build your application, you will receive these two warnings:

   ```
   sfp.c(441) : CC78K0 warning W0837: Output assembler source file,
   not object file

   sfp.c(440) : CC78K0 warning W0915: Asm statement found. skip to
   jump optimize this function 'CallA2ModeFunction'
   ```

   These are normal and need no correction.

5.1 Mode Control Functions

The following functions are used to control the operating modes:

- **EnterA1Mode()** carries out the secure procedure for the MCU to enter A1 mode
- **ReturnToNormalMode()** returns the MCU from A1 mode to normal operation

5.2 Entry RAM and Data Buffer Address Functions

These functions let the flash self-programming functions know the address of the arrays used for the entry RAM and data buffer. This must be done only once before you enter A1 mode, unless you want to change the array used for the entry RAM or data buffer.

- **SetEntryRamAddress(myEntryRam)** sets the address of the array you are using as your entry RAM
- **SetDataBufferAddress(myDataBuffer)** sets the address of the array used for the data buffer

In addition, these additional functions are provided for completeness:

- **GetEntryRamAddress();**
- **GetDataBufferAddress();**

5.3 Flash Self-Programming Functions

When the MCU is in A1 mode, these functions can be used to call hidden ROM functions in A2 mode to perform flash self-programming operations:

5.3.1 SFP Functions

- **SFP_ModeCheck()** checks that the MCU is in correct mode and FLMD0 pin is high
- **SFP_Initialise(MICRO_CLOCK_FRQ)** initializes and sets the clock frequency
- **SFP_BlockBlankCheck(blocknumber)** checks specified block to see if it is blank
- **SFP_BlockErase(blocknumber)** erases a specified block
- **SFP_WordWrite(start_address, number_of_words)** writes a number of words from the data buffer to flash memory
- **SFP_BlockVerify(blocknumber)** verifies a specified block

5.3.2 Additional SFP Functions

- **SFP_GetInformation()** gets information on write/erase enable/disable for external programmer
- **SFP_SetInformation()** sets or clears bits for bootswapping operation

6. Flash Self-Programming

The following description and flowcharts show a typical flash self-programming sequence.

Figure 4. Steps 1–7 of Self-flash Programming Sequence

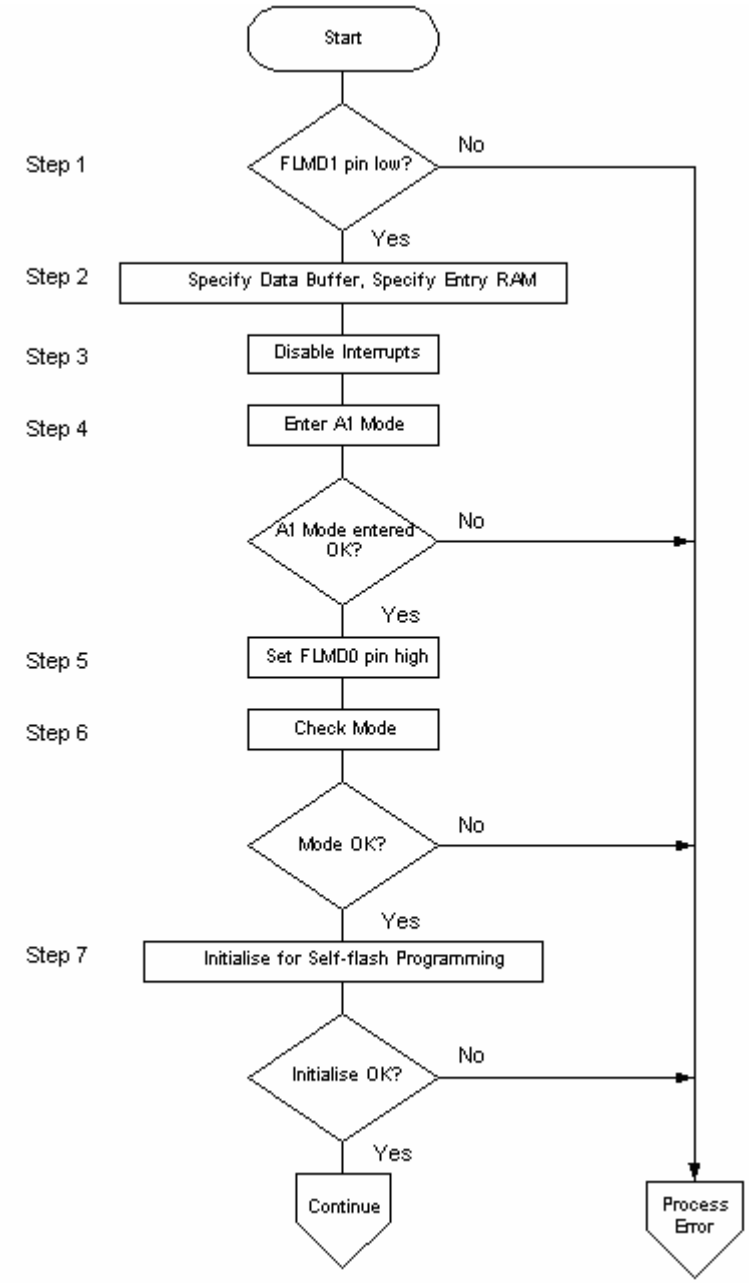


Figure 5. Steps 8–10 of Flash Self-Programming Sequence

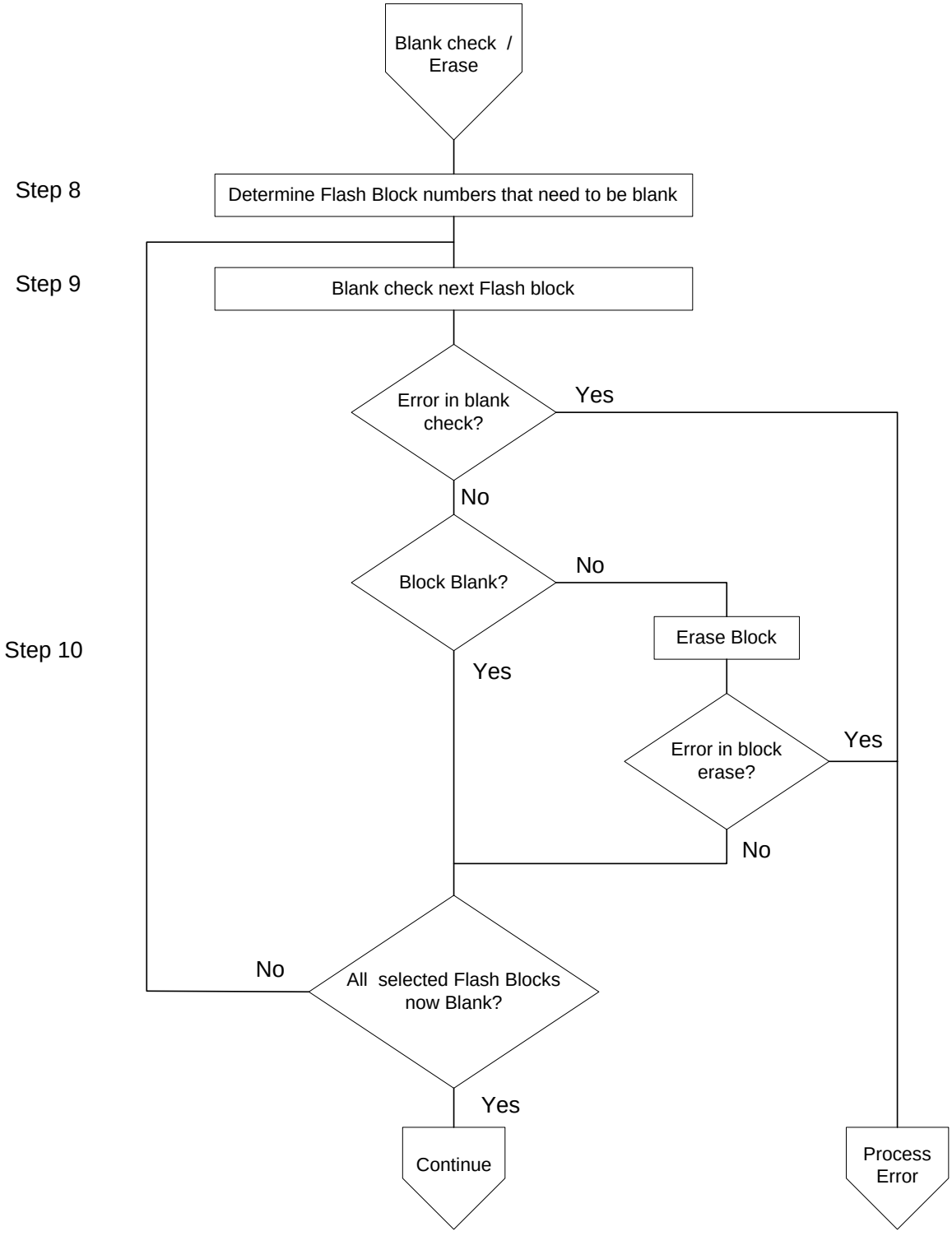


Figure 6. Steps 11–14 of Flash Self-Programming Sequence

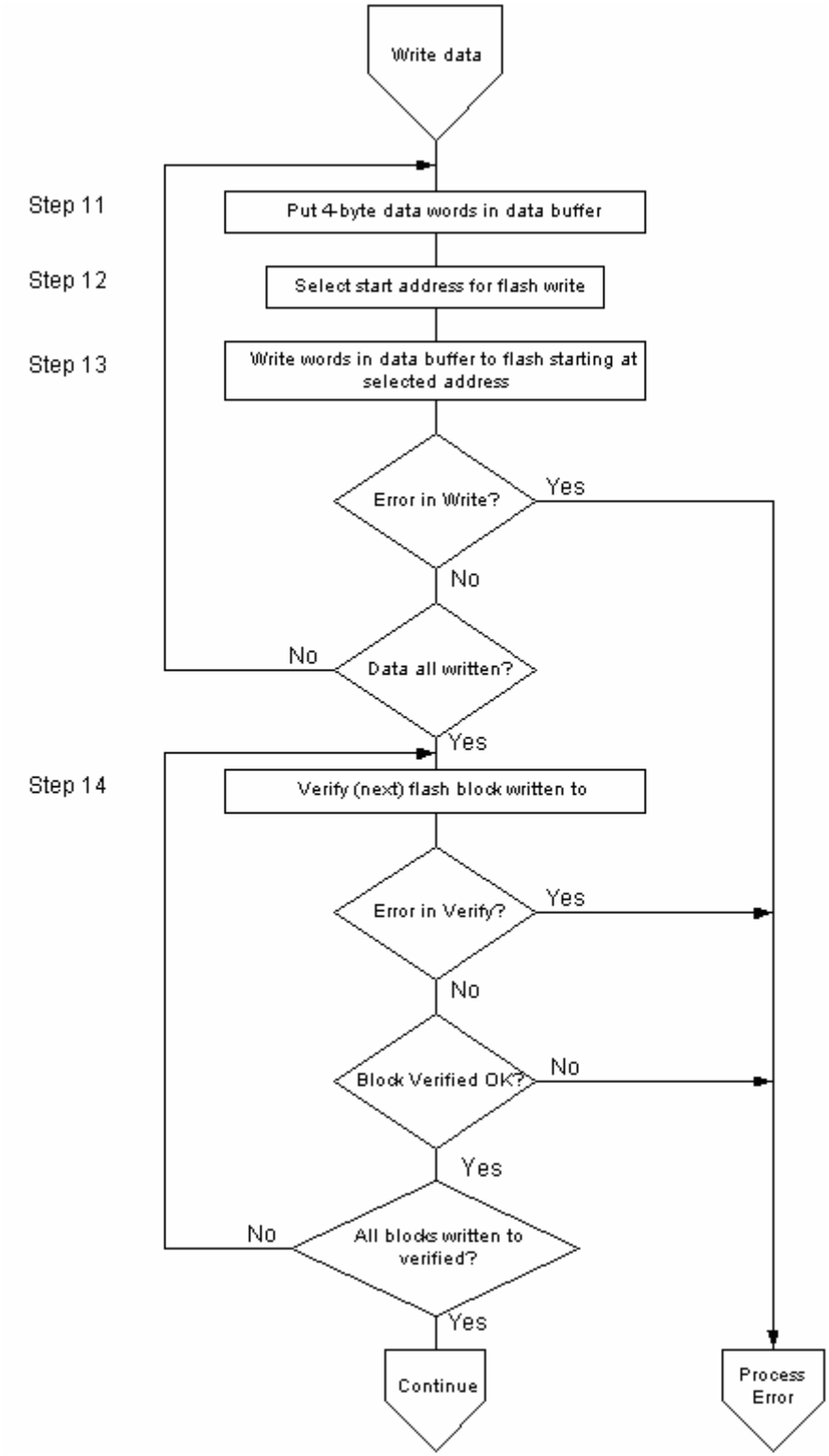
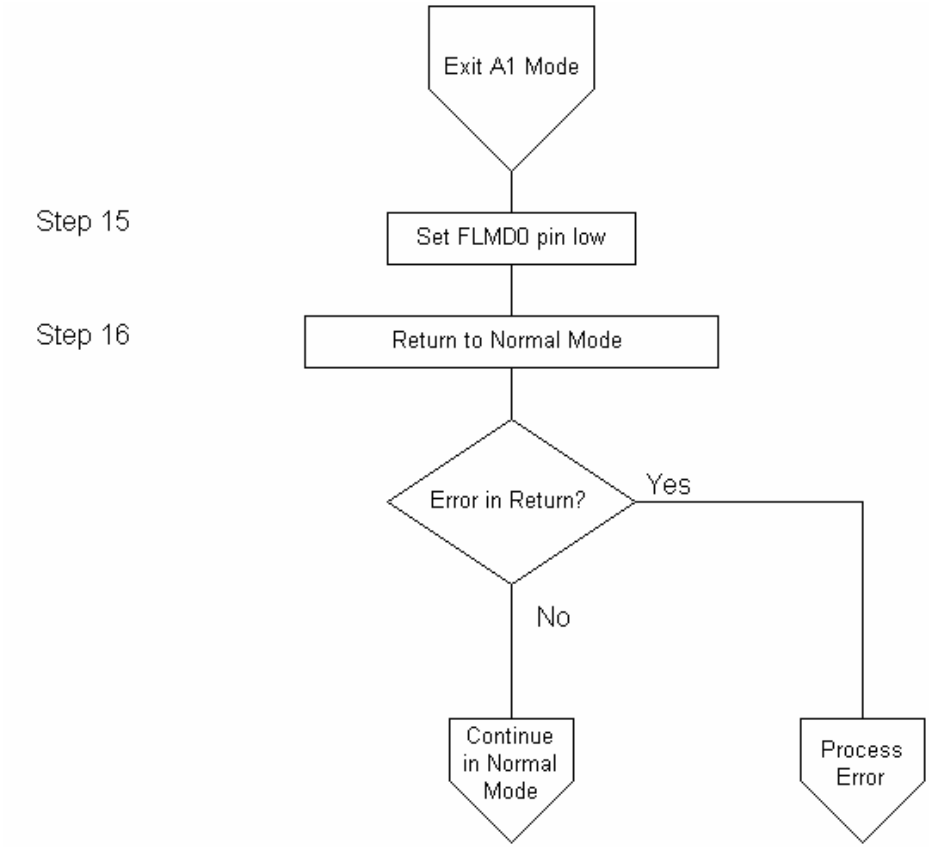


Figure 7. Steps 15–16 of Flash Self-Programming Sequence



6.1 Step 1. Verify That the FLMD1 Pin is Low

This is important, as you must set the FLMD0 pin high after the MCU enters A1 mode. If FLMD0, FLMD1 and RESET are all high, the MCU enters a test mode. RESET is normally high and FLMD0 must be high for flash self-programming, so it is important to ensure that FLMD1 is low.

Example:

```
#define FLMD1_PIN P1.7                  // define pin for FLMD1
if(FLMD1_PIN!=0) ProcessError();
```

6.2 Step 2. Set the Addresses for Entry RAM and the Data Buffer

6.2.1 Setting the Entry RAM Address

The entry RAM can be anywhere in high-speed RAM or extension RAM. The entry RAM must be 48 bytes. The **SetSfpEntryRamAdress()** function lets the self-flash programming functions know the address of your application's entry RAM. Commonly this would only be set once, upon entering A1 mode, as the entry RAM address is remembered. However, it also can be set at any time to a different address before using any of the flash self-programming functions.

Example:

```
#define ER_SIZE 48      // 48 bytes required for entry RAM
unsigned char myentryram[ER_SIZE]; // users Entry RAM
SetSfpEntryRamAddress(myentryram);
```

6.2.2 Setting the Data Buffer Address

The data buffer can be anywhere in high-speed RAM or extension RAM. Data buffer size is from 4 bytes to 256 bytes. Data is written in four-byte words, so the data buffer contains 1 to 64 words. It is best to round up the data buffer size to the nearest number of whole words. In general, making the data buffer three bytes larger than the maximum amount of bytes you wish to write can ensure this.

The **SetSfpDataBufferAdresss()** function lets the self-flash programming functions know the address of your application's data buffer. Commonly this would only be set once on entering A1 mode, as the data buffer address is remembered. However it can be also be set at any time to a different address before using any of the flash self-programming functions.

Example:

```
#define DB_SIZE SAMPLE_SIZE+3           // data buffer size
unsigned char mydatabuffer[DB_SIZE];    // users Data Buffer
SetSfpDataBufferAddress(mydatabuffer);
```

6.3 Step 3. Disable Interrupts

Interrupts should be disabled before entering A1 mode. The secure process to enter A1 mode cannot be interrupted or it will fail. It is generally advisable to leave interrupts disabled in A1 mode, although they may be used with caution. The A2 mode flash self-programming functions must *never* be interrupted.

Make sure you have the pragma directive at the start of your file:

Example:

```
#pragma DI
Func() {
DI();
}
```

6.4 Step 4. Enter A1 Mode

The self-flash programming functions in the hidden ROM can only be accessed when the MCU is in A1 mode.

Example:

```
returnValue=EnterA1Mode();
if(returnValue!=0) ProcessError();
```

Check the returned value to see the result:

- 0: successful in entering A1 mode
- 1: error in entering A1 mode – bit FPERR of the PFS register is set
  Correct sequence was not followed or sequence was interrupted.

6.5 Step 5. Set the FLMD0 Pin High

The FLMD0 pin must be high for the flash self-programming function to work

1. Set the port pin controlling the FLMD0 level to output high.
2. Refer to Figure 3 for the circuit configuration. For example, if Port 3.0 controls the level of the FLMD0 pin, then use these #define statements:

```
#define SET_FLMD0_CONTROL_TO_OUTPUT PM3.0=0 // define for FLMD0
control pin
#define SET_FLMD0_CONTROL_TO_INPUT PM3.0=1       // define for
FLMD0 control pin
#define SET_FLMD0_CONTROL_HIGH P3.0=1
     // set FLMD0 control pin high
```

and then these statements:

```
SET_FLMD0_CONTROL_TO_OUTPUT;
SET_FLMD0_CONTROL_HIGH;
```

6.6 Step 6. Check the Mode Using the SFP_CheckMode() Flash Self-Programming Function

Example:

```
returnValue=SFP_ModeCheck();
if(returnValue!=0) ProcessError();
```

Check the return value to see the result:

- 0: successful completion of function
- 1: error in mode – bit FWEPR of the FLPMC register is 0 (FLMD0 pin is low)

6.7 Step 7. Call the Flash Self-Programming Initialization Function

This function sets the clock frequency for flash self-programming. Use a #define statement to specify the value of the MCU's clock frequency in hertz (Hz).

Example:

```
#define MICRO_CLOCK_FRQ 8000000       // microcontroller crystal
frequency in Hz
```

Use a statement of the form:

```
returnValue=SFP_Initialise(MICRO_CLOCK_FRQ);
if(returnValue!=0) ProcessError();
```

Then check the return value to see the result:

```
ReturnValue:
00H: successful completion of function
05H:  clock frequency is outside allowable range
```

6.8 Step 8. Check Which Memory Blocks Must Be Blank

This step entails determining which blocks you require to be blank. If you are using a bootloader to download a new application program, you will probably require that all blocks outside the boot area to be blank. Alternatively, you may check the addressing and just blank check the blocks that will be written to. However, if you have written data to a block previously and now wish to continue, it is not necessary to blank check the block. You just need to keep track of the blank area in the block. This is a common requirement as the memory block is 2 KB, whereas the maximum amount of data that can be written to a block at one time is 256 bytes (maximum data buffer size).

This information is needed for Steps 9 and 10, which blank check / erase the required blocks.

6.9 Step 9. Verify That Selected Memory Block is Blank

If you require a block to be blank, it is best to execute a **Blank Check** command before executing an **Erase** command. The 78K0/KX1+ MCUs have a finite limit to the amount of the times they can be erased/written, and taking this step cuts down on the needless erasing of already-blank blocks. Refer to the MCU's specifications for more details.

To check a memory block is blank, use the type of statement shown below:

Example:

```
returnValue=SFP_BlockBlankCheck(blocknumber);
if(returnValue!=0)
     {
if(returnValue!=0x1b)
{
ProcessError();
}
else
{
….// block is not blank …..go on to erase block
}
}
else
{
…. // block is blank
}
```

Where block_number is the number of one of the 2 KB memory blocks, numbered 0 to 29, between address 0000H and F000H.

Then check the return value to see the result:

```
ReturnValue:
00H: successful completion of function (block is blank)
05H: parameter error
1BH: blank check error (block is not blank)
```

6.10 Step 10. Erase the Selected Memory Block

This step can be skipped if you previously determined that the block is already blank or if you only want to write to a known blank area of a partially written block.

To erase a memory block, use this type of statement:

Example:

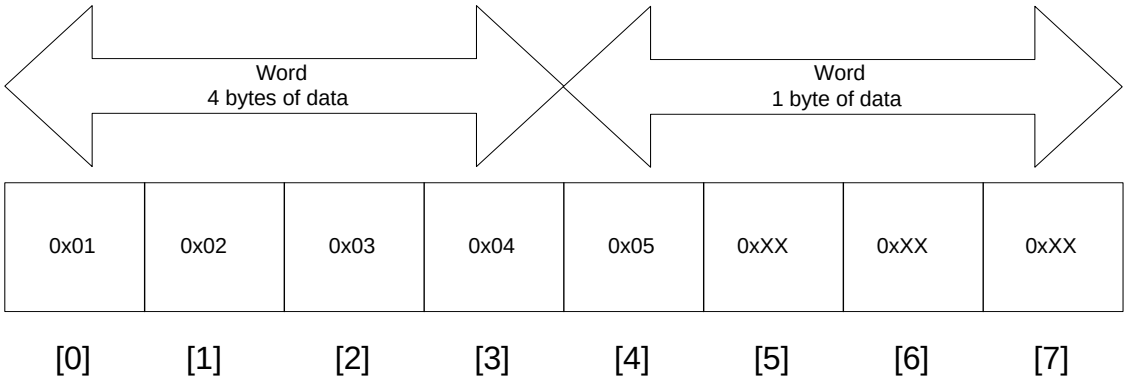
```
returnValue=SFP_BlockErase(blocknumber);
if(returnValue!=0) ProcessError();
```

Where **block_number** refers to one of the 2 KB memory blocks, numbered 0 to 29, between addresses 0000H and EFFFH. Then check the return value to see the result:

```
ReturnValue:
00H: successful completion of function (block is erased)
05H: parameter error
1AH: erasing error (block could not be erased)
```

6.11 Step 11. Put the Data to be Written in the Data Buffer

Figure 8. Bytes Written From the Data Buffer



Data is always written in four-byte words. In Figure 5, five bytes of data are written so that the remaining three bytes in the second word are written with whatever value (0xXX) happens to be in the data buffer.

If you have a number of bytes to write that is not evenly divisible by four, you may wish to pad the data with a known value to make a whole number of words. If you don't wish to pad out your data, then the write function uses whatever values happen to be in the data buffer.

You will also need to specify the number of words to be written. If you divide the number of bytes by four, you will get the number of words that contain four bytes of data. If you find a remainder when you divide by four, you must increase this number of words by one, to make sure the remaining bytes are also written.

Example:

```
#define SAMPLE_SIZE 5                  // size of sample data
#define DB_SIZE SAMPLE_SIZE+3          // data buffer size
#define PAD_BYTE 0x00                  // byte used to pad data
const unsigned char sampledata[SAMPLE_SIZE]={1,2,3,4,5};
for(i=0;i<DB_SIZE;i++)
      {
mydatabuffer[i]=PAD_BYTE;              // fill buffer with pad
byte
}
      for(i=0;i<SAMPLE_SIZE;i++)       // put data in data buffer
{
mydatabuffer[i]=sampledata[i];
}
number_of_bytes=SAMPLE_SIZE;           // get number of bytes to
be written
number_of_words=number_of_bytes/4;     // calculate number of
words
if((number_of_bytes%4)!=0)number_of_words++;
```

6.12 Step 12. Select Starting Address for Flash Write Operation

This address is the address where writing will start using the byte at the first location in the data buffer. The address may be specified by the user, or taken from a hexadecimal file that is being downloaded to the MCU. A three-byte address is used although the 78K1+ has a 64 KB memory space.

Example:

```
#define FIRST_BLOCK_ADDRESS 0x00002000;          // define
address of 1st block
unsigned long write_start_address;
write_start_address=FIRST_BLOCK_ADDRESS;
```

6.13 Step 13. Write Data to Flash Memory

To write to flash memory, use a statement of the form:

Example:

```
returnValue=SFP_WordWrite(write_start_address, number_of_words);
if(returnValue!=0) ProcessError();
```

Where write_start_address is the address where the function starts writing and number_of_words is the number of four-byte words to be written from the data buffer. Then check the return value to see the result:

```
ReturnValue:
```

```
00H: successful completion of function (write of data successful)
05H: parameter error
18H: FLMD0 error
1CH: write error (location(s) could not be written to)
```

6.14 Step 14: Verify a memory block

This function internally verifies a memory block for proper operation. You may wish to verify a memory block either before or after erasing or writing it. To verify a memory block, use a statement of the form:

Example:

```
returnValue=SFP_BlockVerify(blocknumber);
if(returnValue!=0) ProcessError();
```

Where block_number is the number of one of the 2 KB memory blocks, numbered 0–29, between address 0000H and F000H.

Then check the return value to see the result:

```
ReturnValue:
00H: successful completion of function
05H: parameter error
1BH: internal verify error
```

6.15 Step 15. Set the FLMD0 Pin Low

When you are finished using the flash self-programming functions, you should set FLMD0 pin low before resetting the MCU or returning to normal operation.

Example:

```
#define SET_FLMD0_CONTROL_LOW P3.0=0        // set FLMD0 control
pin low
```

Then use this statement: SET_FLMD0_CONTROL_LOW;

6.16 Step 16. Return to Normal Operation

When you are finished using the flash self-programming functions, you shouldreset the MCU or return to normal operation.

To return to normal operation, use a statement of the form:

Example:

```
returnValue=ReturnToNormalMode();
if(returnValue!=0) ProcessError();
```

Check the return value to see the result:

```
ReturnValue:
0: successful in returning to Normal Mode
1: error in returning to Normal Mode – bit FPERR of the PFS
register is set
Correct sequence was not followed or sequence was interrupted.
```

7. Sample Code

```
/*****************************************************************************
*
* FILE                                                  : sfpsteps.c
* DATE                                                : October 20, 2004
* DESCRIPTION  : Program to illustrate steps in self-flash programming
* CPU TYPE                          : NEC 78K0/KF1+ - 78F0148H
*
* Notes:              :
*                           This program is intended as an illustration of the typical
*                           steps used in the self-flash programming procedure.
*                           It writes SAMPLE_SIZE bytes of data to the
*                           first locations of the flash blocks from FIRST_BLOCK
*                           to LAST_BLOCK. This program does not get data into the
*                           microcontoller. The data used is simply a constant array
*                           in memory.
*
*                           There is no error handling in this example. A dummy routine
*                           ProcessError() is called if an error is detected.
*
*****************************************************************************/

#pragma sfr
#pragma NOP
#pragma DI


/*=====================================
; include files
;=====================================*/

#include "sfp.h"                    // include self-flash programming routines

/*=====================================
; defines
;=====================================*/
#define BLOCK_SIZE 0x00000800;                              // define 2K block size
#define FIRST_BLOCK_ADDRESS 0x00002000;             // define address of 1st block
#define FIRST_BLOCK 4                                       // define 1st block
#define LAST_BLOCK 29                                       // define last block

#define FLMD1_PIN P1.7                                      // define pin for FLMD1
#define SET_FLMD0_CONTROL_TO_OUTPUT PM3.0=0  // define for FLMD0 control pin
#define SET_FLMD0_CONTROL_TO_INPUT PM3.0=1   // define for FLMD0 control pin
#define SET_FLMD0_CONTROL_HIGH P3.0=1        // set FLMD0 control pin high
#define SET_FLMD0_CONTROL_LOW P3.0=0         // set FLMD0 control pin low

#define MICRO_CLOCK_FRQ 8000000                             // microcontroller clock frequency

#define SAMPLE_SIZE 5                                       // size of sample data
#define DB_SIZE SAMPLE_SIZE+3                       // data buffer size
#define ER_SIZE 48                                                  // 48 bytes required for entry
       ram

#define PAD_BYTE 0x00                                       // byte used to pad data

/*=====================================
; function prototypes
;=====================================*/

void ProcessError(Void);                                    // dummy  - no error processing

/*=====================================
; ROM constants
;=====================================*/

const unsigned char sampledata[SAMPLE_SIZE]={1,2,3,4,5};


/*=====================================
```

```
; variables
;====================================*/

unsigned char blocknumber;
unsigned char i,returnValue,number_of_bytes,number_of_words;
unsigned char myentryram[ER_SIZE];          // users entry RAM
unsigned char mydatabuffer[DB_SIZE];  // users Data Buffer
unsigned long write_start_address;

/*************************************************/
void main (void){

        IMS = 0x0CF;          // set internal memory size for 780148H (60KB)
        IXS=0x0A;                     // set expansion RAM size for 780148H (1KB)

        while(OSTC != 0x1f) ;  // wait for oscillator to stabilize
        MCM0 = 1;                             // switch to oscillator

        WDTM = 0x77;                  // no watchdog


//*********************************************
// Step 1: Check that the FLMD1 pin is low
//*********************************************

        if(FLMD1_PIN!=0) ProcessError();

//*********************************************
// Step 2: Set the entry RAM and Data Buffer addresses
//*********************************************

        SetSfpDataBufferAddress(mydatabuffer);
        SetSfpEntryRamAddress(myentryram);

//*********************************************
// Step 3: Disable interrupts
//*********************************************

        DI();

//*********************************************
// Step 4: Enter A1 Mode
//*********************************************

        returnValue=EnterA1Mode();
        if(returnValue!=0) ProcessError();

//*********************************************
// Step 5: Set FLMD0 pin high
//*********************************************

        SET_FLMD0_CONTROL_TO_OUTPUT;
        SET_FLMD0_CONTROL_HIGH;

//*********************************************
// Step 6: Check the Mode
//*********************************************

        returnValue=SFP_ModeCheck();
        if(returnValue!=0) ProcessError();

//*********************************************
// Step 7: Initialize for self-flash programming
//*********************************************

        returnValue=SFP_Initialise(MICRO_CLOCK_FRQ);
        if(returnValue!=0) ProcessError();
```

```
//*********************************************
// Step 8: Determine blocks to be blank checked/erased
//*********************************************

               // block numbers given by defines

for(blocknumber=FIRST_BLOCK;blocknumber<LAST_BLOCK+1;blocknumber++)
               {
//*********************************************
// Step 9: Blank check a block
//*********************************************

                        returnValue=SFP_BlockBlankCheck(blocknumber);
                        if(returnValue!=0)
                        {
                               if(returnValue!=0x1b)
                               {
                                      ProcessError();
                               }
                               else
                               {
//*********************************************
// Step 10: Erase a block
//*********************************************

                                      returnValue=SFP_BlockErase(blocknumber);
                                      if(returnValue!=0) ProcessError();
                               }
                        }
               }
//*********************************************
// Step 11: Put data in Data Buffer
//*********************************************
       for(i=0;i<DB_SIZE;i++)
       {
              mydatabuffer[i]=PAD_BYTE;            // fill buffer with pad byte
       }
       for(i=0;i<SAMPLE_SIZE;i++)                  // put data in Data Buffer
       {
              mydatabuffer[i]=sampledata[i];
       }
       number_of_bytes=SAMPLE_SIZE;         // get number of bytes to be written
       number_of_words=number_of_bytes/4;   // calculate number of words
       if((number_of_bytes%4)!=0)number_of_words++;

//*********************************************
// Step 12: Select start address for flash write
//*********************************************
               write_start_address=FIRST_BLOCK_ADDRESS;

//*********************************************
// Step 13: Write data
//*********************************************

               for(blocknumber=FIRST_BLOCK;blocknumber<LAST_BLOCK+1;blocknumber++)
               {

                      returnValue=SFP_WordWrite(write_start_address,number_of_words);
                      if(returnValue!=0) ProcessError();

//*********************************************
// Step 14: Verify block written to
//*********************************************

                      returnValue=SFP_BlockVerify(blocknumber);
                      if(returnValue!=0) ProcessError();

                      write_start_address+=BLOCK_SIZE; // add block size
               }
```

```
//*********************************************
// Step 15: Set FLMD0 pin low
//*********************************************

        SET_FLMD0_CONTROL_LOW;

//*********************************************
// Step 16: Return to Normal Mode
//*********************************************

        returnValue=ReturnToNormalMode();
        if(returnValue!=0) ProcessError();

//*********************************************
// End of steps
//*********************************************
        NOP();
        while(1);       // endless loop

}
//*********************************************
// Dummy routine for error processing
//*********************************************

void ProcessError(void)
{
        NOP();
        while(1);       // endless loop
}
```

Appendix A

Flash Self-Programming Routines

```
/*****************************************************************************
*
* FILE                                                    : sfp.h
* DATE                                                 : December 13, 2004
* DESCRIPTION  : Self-flash programming header file
* CPU TYPE                          : 78K0/KF1+ (uPD78F0148H)
*
* Notes:
*
*****************************************************************************/
#ifndef _SFP_H
#define _SFP_H


// defines for self flash programming function numbers for 78F0148H
#define INITIALIZATION              0x00
#define BLOCK_ERASE           0x03
#define WORD_WRITE                  0x04
#define BLOCK_VERIFY          0x06
#define BLOCK_BLANK_CHECK     0x08
#define GET_INFORMATION       0x09
#define SET_INFORMATION       0x0a
#define MODE_CHECK                  0x0e
#define EEPROM_WRITE          0x17

// defines for other functions
#define ENTER_A1                    0xa1
#define RETURN_NORMAL         0x5f

// return value defines
#define NOT_BLANK                   0x1b


// function prototypes
void SetSfpDataBufferAddress(unsigned char *data_buffer);
void SetSfpEntryRamAddress(unsigned char *entry_ram);
unsigned char EnterA1Mode(void);
unsigned char ReturnToNormalMode(void);


// function prototypes for interfacing to
// A2 Mode self flash programming routines in hidden ROM
unsigned char SFP_Initialise(unsigned long clock_frq);
unsigned char SFP_BlockErase(unsigned char block_number);
unsigned char SFP_WordWrite(unsigned long block_start_address, unsigned char num_of_words);
unsigned char SFP_BlockVerify(unsigned char block_number);
unsigned char SFP_BlockBlankCheck(unsigned char block_number);
unsigned char SFP_GetInformation(unsigned char option_number, unsigned char block_number);
unsigned char SFP_SetInformation(unsigned char information_byte);
unsigned char SFP_ModeCheck(void);
void CallA2ModeFunction(unsigned char function_number);


#endif  /* _SFP_H */
/*****************************************************************************
*
* FILE                                                        : sfp.c
* DATE                                                     : December 13, 2004
* DESCRIPTION          : Self flash programming file
* CPU TYPE                                  : 78K0/KF1+  (uPD78F0148H)
*
* Notes:               :
*
*****************************************************************************/
```

```
#define SFP_C
#pragma sfr
#pragma NOP                          /* key word for NOP instruction */
#pragma asm
#include "sfp.h"

unsigned char *sfpDataBuffer;
unsigned char *sfpEntryRam;

/*****************************************************************************
*
* Function:         SFP_Initialise
* Parameters:         Clock Frequency
* Returns:          Function return value
* Date:             August 29, 2004
* Description:      Sets clock frequency
*
* Notes:                    Function number:      00H
*
*                                  Return value:         00H normal completion
*                                                              05H parameter error
*
*                                  This function is also used to set the
*                                  Data Buffer Address in entry RAM locations 08 and 09
*                                  This should not need to be done again for the other
*                                  self-flash programming functions
*
*****************************************************************************/

unsigned char SFP_Initialise(unsigned long clock_frq)
{
       unsigned int temp_int;

       // put address of sfpDataBuffer in entryram
       temp_int=(unsigned int)sfpDataBuffer;              // convert pointer to integer
       sfpEntryRam[9]=(unsigned char) (temp_int>>8);       // store ms byte of data buffer
address
       sfpEntryRam[8]=(unsigned char) (temp_int);          // store ls byte of data buffer
address


       // put micro clock frequency in sfpDataBuffer
       sfpDataBuffer[3]=(unsigned char)(clock_frq>>24);
       sfpDataBuffer[2]=(unsigned char)(clock_frq>>16);
       sfpDataBuffer[1]=(unsigned char)(clock_frq>>8);
       sfpDataBuffer[0]=(unsigned char)(clock_frq);


       CallA2ModeFunction(INITIALIZATION);  // call A2 mode function
       return(sfpEntryRam[0]);                            // return value in first location of
entry RAM

}
/*****************************************************************************
*
* Function:         SFP_BlockErase
* Parameters:         Block number
* Returns:          Function return value
* Date:             September 3, 2004
* Description:      Erases specified block
*
* Notes:                    Function number:      03H
*
*                                  Return value:         00H normal completion
*                                                              05H parameter error
*                                                              1AH erasing error
*
*****************************************************************************/
```

```
unsigned char SFP_BlockErase(unsigned char block_number)
{
       sfpEntryRam[7]=block_number;

       CallA2ModeFunction(BLOCK_ERASE);     // call A2 mode function
       return(sfpEntryRam[0]);              // return value in first location of entry RAM

}
/*****************************************************************************
*
* Function:         SFP_WordWrite
* Parameters:         Flash start address, number of words
* Returns:          Function return value
* Date:             September 3, 2004
* Description:      Writes word(s) starting at specified start address
*
* Notes:                     Function number:      04H
*
*                                  Return value:         00H normal completion
*                                                        05H parameter error
*                                                        18H FLMD0 error
*                                                        1CH    write error
*
*****************************************************************************/

unsigned char SFP_WordWrite(unsigned long write_start_address, unsigned char num_of_words)
{
       unsigned int temp_int;


       // put write start address in entryram
       sfpEntryRam[3]=(unsigned char)(write_start_address>>16);
       sfpEntryRam[2]=(unsigned char)(write_start_address>>8);
       sfpEntryRam[1]=(unsigned char)(write_start_address);


       // put number of words to be written in entryram
       sfpEntryRam[7]=num_of_words;

       CallA2ModeFunction(WORD_WRITE);      // call A2 mode function
       return(sfpEntryRam[0]);                              // return value in first location of
entry RAM

}


/*****************************************************************************
*
* Function:         SFP_BlockVerify
* Parameters:         Block number
* Returns:          Function return value
* Date:             September 3, 2004
* Description:      Internally verifies specified block
*
* Notes:                     Function number:      06H
*
*                                  Return value:         00H normal completion
*                                                        05H parameter error
*                                                        1BH internal
verification error
*
*****************************************************************************/

unsigned char SFP_BlockVerify(unsigned char block_number)
{
       sfpEntryRam[7]=block_number;  // put block number in entry RAM

       CallA2ModeFunction(BLOCK_VERIFY);    // call A2 mode function
       return(sfpEntryRam[0]);                              // return value in first location of
entry RAM
}

/*****************************************************************************
```

```
*
* Function:         SFP_BlockBlankCheck
* Parameters:         Block number
* Returns:          Function return value
* Date:             September 3, 2004
* Description:      Checks if specified block is blank
*
* Notes:                      Function number:      08H
*
*                                     Return value:         00H normal completion
*                                                           05H parameter error
*                                                           1BH blank check error
*
*********************************************************************************/

unsigned char SFP_BlockBlankCheck(unsigned char block_number)
{
       sfpEntryRam[7]=block_number;  // put block number in entry RAM

       CallA2ModeFunction(BLOCK_BLANK_CHECK);       // call A2 mode function
       return(sfpEntryRam[0]);                              // return value in first location of
entry RAM
}

/*********************************************************************************
*
* Function:         SFP_GetInformation
* Parameters:         Option number, block number
* Returns:          Function return value
* Date:             September 7, 2004
* Description:      Gets information specified by option
*
* Notes:                      Function number:      0EH
*
*                                     Option value:
*                                                           03H security flag
information (block number paramter ignored)
*                                                           04H boot flag
information (block number paramter ignored)
*                                                           05H end address of
specified block
*
*                                     Return value:         00H normal completion
*                                                           05H paramter error
*
*********************************************************************************/

unsigned char SFP_GetInformation(unsigned char option_number, unsigned char block_number)
{
       sfpEntryRam[7]=option_number; // put option number in entry ram
       sfpEntryRam[1]=block_number;  // put block number in entry ram

       CallA2ModeFunction(GET_INFORMATION);  // call A2 mode function
       return(sfpEntryRam[0]);                              // return value in first location of
entry RAM
}

/*********************************************************************************
*
* Function:         SFP_SetInformation
* Parameters:         Information byte to be set
* Returns:          Function return value
* Date:             September 7, 2004
* Description:      Sets information byte
*
* Notes:                      Function number:      0AH
*
*                                     Return value:         00H normal completion
*                                                           05H parameter error
*                                                           18H FLMD0 error
*                                                           1CH    write error
```

```
*                                                                              1DH internal
verification error
*                                                                              1EH     blank error
*
*******************************************************************************/

unsigned char SFP_SetInformation(unsigned char information_byte)
{
       information_byte &= 0x0f;                    // clear upper 4 bits of information byte
       sfpDataBuffer[0]=information_byte;    // store information byte in Data Buffer

       CallA2ModeFunction(SET_INFORMATION);  // call A2 mode function
       return(sfpEntryRam[0]);                              // return value in first location of
entry RAM
}
/*******************************************************************************
*
* Function:         SFP_ModeCheck
* Parameters:         none
* Returns:          Function return value
* Date:             September 7, 2004
* Description:      Internally verifies specified block
*
* Notes:                     Function number:       0EH
*
*                                  Return value:          00H normal completion (FLMD0 pin is
high)
*                                                                              01H error (FLMD0 pin
is low)
*
*******************************************************************************/

unsigned char SFP_ModeCheck(void)
{
       CallA2ModeFunction(MODE_CHECK);       // call A2 mode function
       return(sfpEntryRam[0]);                      // return value in first location of entry
RAM
}


/*******************************************************************************
*
* Function:         SetSfpDataBufferAddress
* Parameters:         pointer to data buffer
* Returns:          nothing
* Date:             September 7, 2004
* Description:      Sets pointer to data buffer to be used in self flash programming
*
* Notes:
*                                  Data buffer size: 4 to 256 bytes (or words??)
*                                  Data buffer location: anywhere in high speed or extension
RAM
*
*******************************************************************************/

void SetSfpDataBufferAddress(unsigned char *data_buffer)
{
       sfpDataBuffer=data_buffer;
}
/*******************************************************************************
*
* Function:         SetSfpEntryRamAddress
* Parameters:         pointer to entry RAM
* Returns:          nothing
* Date:             September 7, 2004
* Description:      Sets pointer to entry ram to be used in self flash programming
*
* Notes:
*                                  Data buffer size: 48 bytes
*                                  Data buffer location: anywhere in high speed or extension
RAM
*
*******************************************************************************/
```

```
void SetSfpEntryRamAddress(unsigned char *entry_ram)
{
       sfpEntryRam=entry_ram;
}

/*****************************************************************************
*
* Function:         EnterA1Mode
* Parameters:         none
* Returns:          flash status register PFS
* Date:             August 27, 2004
* Description:      Enters A1 mode
*
* Notes:
*                     PFCMD   Flash Protect Command Register
*                     FLPMC   Flash Programming Mode Control Register
*                     PFS           Flash Status Register
*
*****************************************************************************/

unsigned char EnterA1Mode(void)
{
       PFCMD=0xa5;                  // required write of 0A5H
       FLPMC=0x05;                  // write value to enter A1 mode
       FLPMC=0xfa;                  // write inverse of value (oxfa is inverse of 0x05)
       FLPMC=0x05;                  // write value to enter A1 mode again
       return PFS;                  // bit 0,  FPERR, is set if there is an error in above
sequence
}
/*****************************************************************************
*
* Function:         ReturnToNormalMode
* Parameters:         none
* Returns:          flash status register PFS
* Date:             September 3, 2004
* Description:      Returns to Normal mode (from A1 mode)
*
* Notes:
*                     PFCMD   Flash Protect Command Register
*                     FLPMC   Flash Programming Mode Control Register
*                     PFS           Flash Status Register
*
*****************************************************************************/

unsigned char ReturnToNormalMode(void)
{
       PFCMD=0xa5;                  // required write of 0A5H
       FLPMC=0x00;                  // write value to enter normal mode
       FLPMC=0xff;                  // write inverse of value (oxff is inverse of 0x00)
       FLPMC=0x00;                  // write value to enter normal mode again
       return PFS;                  // bit 0,  FPERR, is set if there is an error in above
sequence
}

/*****************************************************************************
*
* Function:         CallA2ModeFunction
* Parameters:         function number, entry RAM address
* Returns:          Register B return value of hidden ROM function
* Date:             September 10, 2004
* Description:      Calls hidden ROM function in A2 Mode
*
*
* Notes:
*                                   To interface to the hidden rom functions
*                                   Function number must be in register c of bank 3
*                                   entry RAM Address must be in register pair hl of bank 3
*
*****************************************************************************/
```

```
void CallA2ModeFunction(unsigned char function_number)
{
#asm
       push ax                              ; save parameter passed in ax register
       sel rb3                              ; select register bank 3
       pop ax                               ; restore saved paramter into ax register of bank 3
       xch a,x                              ; move function_number from x to a
       mov c,a                              ; move function_number from a to c
       movw ax,!_sfpEntryRam  ; move address of entry RAM into ax
       movw hl,ax                           ; move address of entry RAM into hl
       call !08100H                 ; call A2 Mode hidden ROM function
#endasm
}
```