NEC

Application Note

78K0/Kx1+ Self-Programming
Bootloader Example

Document No. U18054EU1V0AN00

Revision History

Date	Revision	Section	Description
04-2006	—	—	First release

Contents

1. Introduction

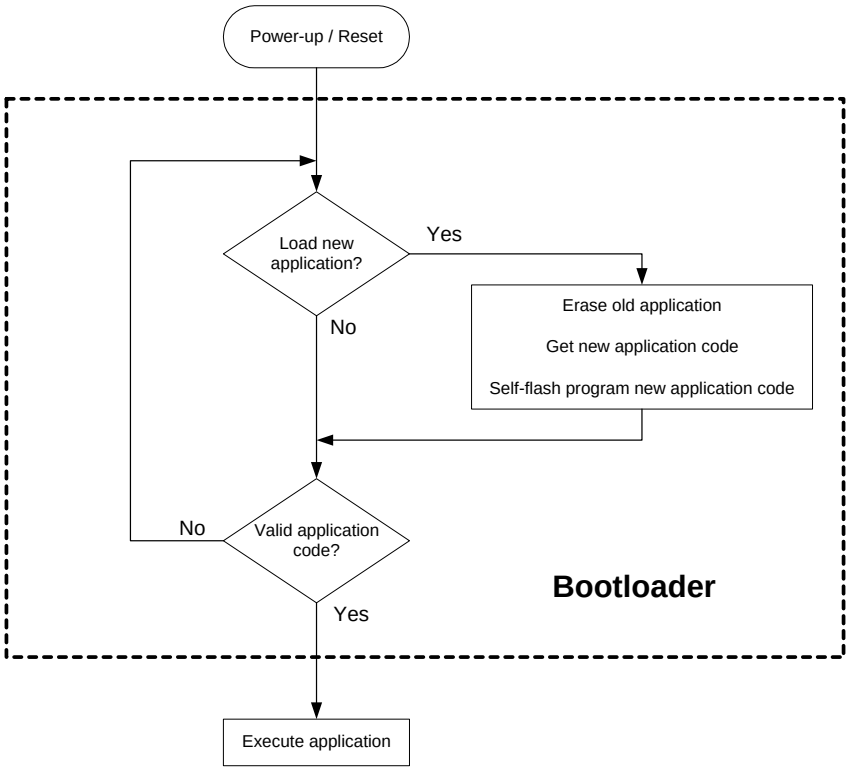
This application note describes how to implement a bootloader using the NEC Electronics µPD78F0148H MCU (MCU). To request the sample code for implementing the self-programming function in other 78K0/Kx1+ MCUs, please contact your local NEC Electronics America representative.

1.1 Definition of "Bootloader"

The "boot" name derives from the lines of code that execute when the MCU starts up or "boots". "Loader" derives from the "loading" and writing of new application code into the MCU's flash memory. An MCU must be capable of flash self-programming to be able to implement a bootloader.

Based on an external signal, a bootloader will either load a new application or proceed to the existing application (Figure 1).

Figure 1. Basic Bootloader



1.2 Advantages

A bootloader gives you the ability to update or replace application code without the use of an external programmer, and also makes it possible to update code remotely over a phone line or Internet connection.

For example, if there were 5,000 MCU-based pay phones in California and the phones needed a firmware update, the phone company's service person could manually reprogram the telephones one at a time using an external programmer, a time-consuming effort, or use a bootloader to reprogram all 5,000 phones remotely from one central location.

1.3 First-Time Programming

Bootloader code generally does *not* come preprogrammed in the MCU. Code sometimes is provided by an MCU supplier and, in other cases, it is written by the MCU user. The code then is programmed into an MCU using an external programmer. Once programmed, the bootloader can by used to load a user's application program without the need for an external programmer. When the application program needs updating, the bootloader can be instructed to load the updated program.

1.4 Updates

Many MCUs require the use of an external programmer when the bootloader itself must be updated, because the bootloader cannot execute code and overwrite itself with new code at the same time. NEC Electronics 78K0/Kx1+ MCUs, however, allow bootloader code to be updated by means of a bootswapping feature.

1.5 Cautions

If there is an interruption or errors in the bootloading process, then there will only be a partial application in memory. The bootloader must guard against trying to execute this invalid code.

1.6 Main Elements

A bootloader consists of five main elements:

- Signal to start the bootloading process
- Signal to execute the bootloader
- Transfer of new code into the MCU
- Flash self-programming of the new code
- Transfer of control to a valid application program

1.7 Start-up Signal

In a situation where hundreds of vending machines were all connected to the Internet and the owner wanted to update their firmware, a signal would have to be generated to trigger the MCUs to start the bootloading process. The signal could be an interrupt, a command byte sent over a serial channel, or something else that would cause the program to reset and run the bootloader code.

1.8 Execution Signal

Upon startup, the MCU loads a new application program or executes an existing one depending the external signal it receives. The signal could be programmed to come from a port pin upon power-up and, depending on whether the signal is high or low, the MCU would load or execute accordingly. The signal could also be based on a character received by the UART, or on the reading taken by the analog-to-digital (A/D) converter. It is up to the user to decide the best method.

1.9 Transferring Code

The data can be transferred over an RS-232, I^2C or serial port or via a parallel port over a number of lines. The user can decide. Since the amount of data transferred typically exceeds the size of the MCU's RAM, there must be some provision to control the flow of the data. For an RS-232 serial port, a slow the baud rate could be used to give the MCU time to process the data and self-program itself without being overrun. Hardware handshaking using clear to send (CTS) and request to send (RTS) lines to control the flow of data would be another option. Another would be to use software handshaking using the XON/XOFF protocol.

New code can be in a format chosen by the user, but it will need to contain addressing information as well as checksums for error processing, typically a standard such as Intel hexadecimal format.

1.10 Flash Self-Programming

Each time an MCU receives a new batch of data, the MCU must self-program itself in the correct flash memory locations. If the locations are not already blank, they must be erased before programming. Also, typically, they must be verified during or after programming.

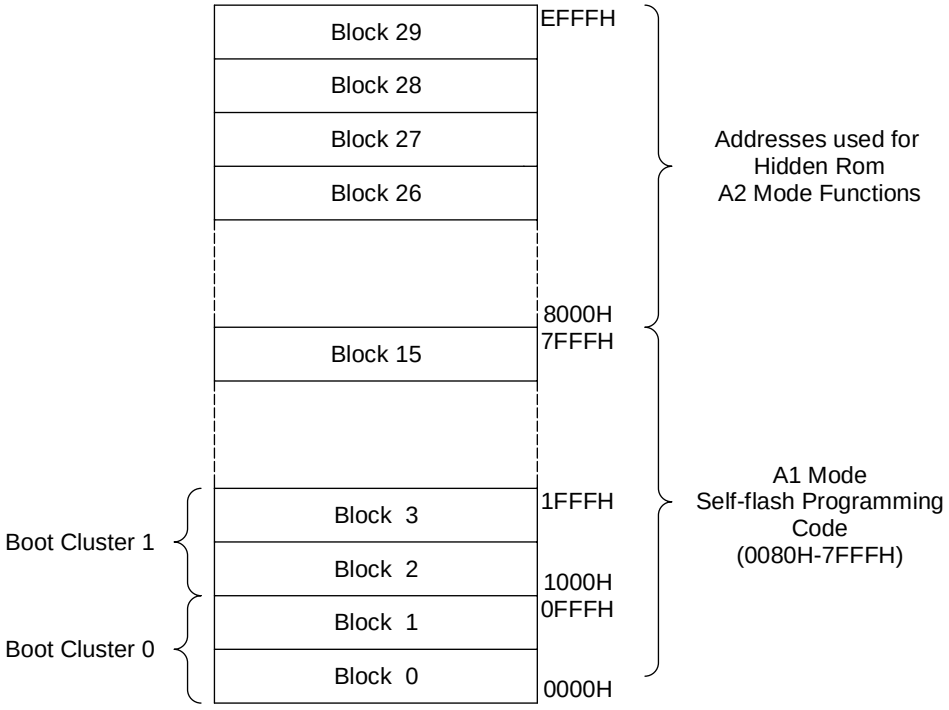
1.11 Transferring Control to a Valid Application Program

Once the new code is received and programmed successfully, the bootloader writes a checksum or other unique byte sequence to a fixed memory location. The bootloader then checks for this valid application checksum or byte sequence and, if it is present, the bootloader transfers control to the application.

2. Flash Self-Programming

To implement a bootloader, an MCU must be capable of flash self-programming. The architecture and mechanisms of flash self-programming in the NEC Electronics 78K0/Kx1+ MCUs are described here. The description will focus on the µPD78F0148H, the MCU used in the bootloader described here.

Figure 2. µPD78F0148H Flash Memory



2.1 Flash Blocks

The 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). See the appendix for a complete list of flash memory block addresses.

2.2 Boot Clusters

The first four blocks of flash memory are divided into two boot clusters of 4 KB, called *boot cluster 0* and *boot cluster 1*. These are used together with the bootswapping feature to allow a bootloader to download a new bootloader that overwrites the old one. For information about the bootswapping procedure, refer to Section 4.

2.3 Flash Words

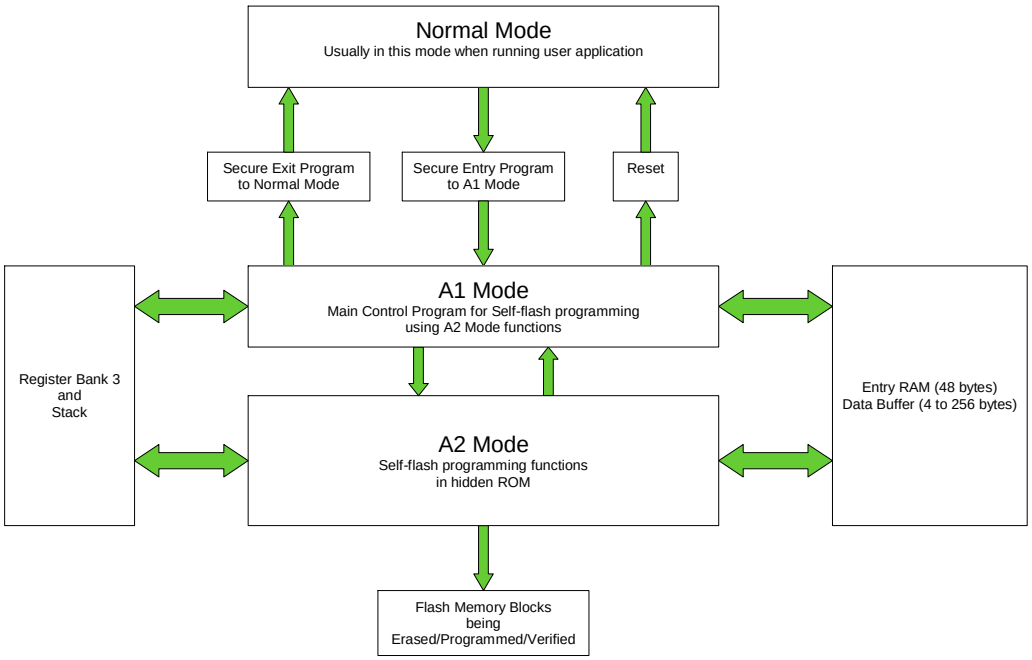
The smallest amount of flash memory that can be written is a word of four bytes. The data buffer used for programming holds 1 to 64 words of data (4–256 bytes).

2.4 Operating Modes

The 78K0/Kx1+ MCUs are capable of flash self-programming and so can implement a bootloader function. To control the flash self-programming process, there are three MCU operating modes:

- Normal operation
- A1 mode
- A2 mode

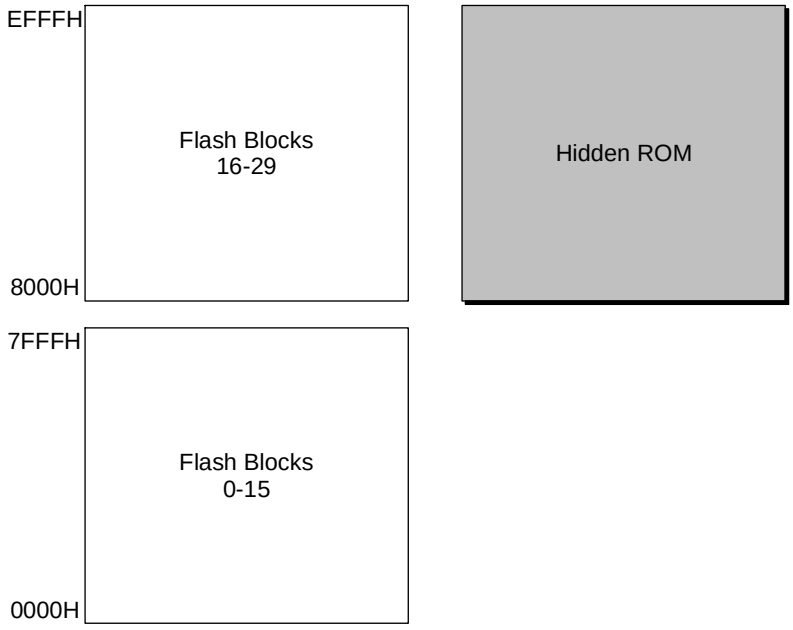
Figure 3. Operating Modes



2.4.1 Normal Operation

Normal operation is, as the name suggests, just the normal operating mode of the MCU as it executes a user program. The MCU cannot self-program its flash memory in this mode. It must first enter A1 mode.

Figure 4. Memory Map in Normal Operation



2.4.2 A1 Mode

The MCU can only enter A1 mode when bytes are written to the **Flash Programming Mode Control** (FLPMC) and **Flash Protect Command** (FPCMD) control registers in a specific sequence. This programming mechanism safeguards the MCU from entering A1 mode inadvertently. Note that the only reason to enter A1 mode is to gain access to the hidden ROM functions for flash self-programming. Once in A1 mode, with the FLMD0 pin set high, the hidden ROM functions for erasing, writing, etc. can be called. Interrupts should be disabled in A1 mode.

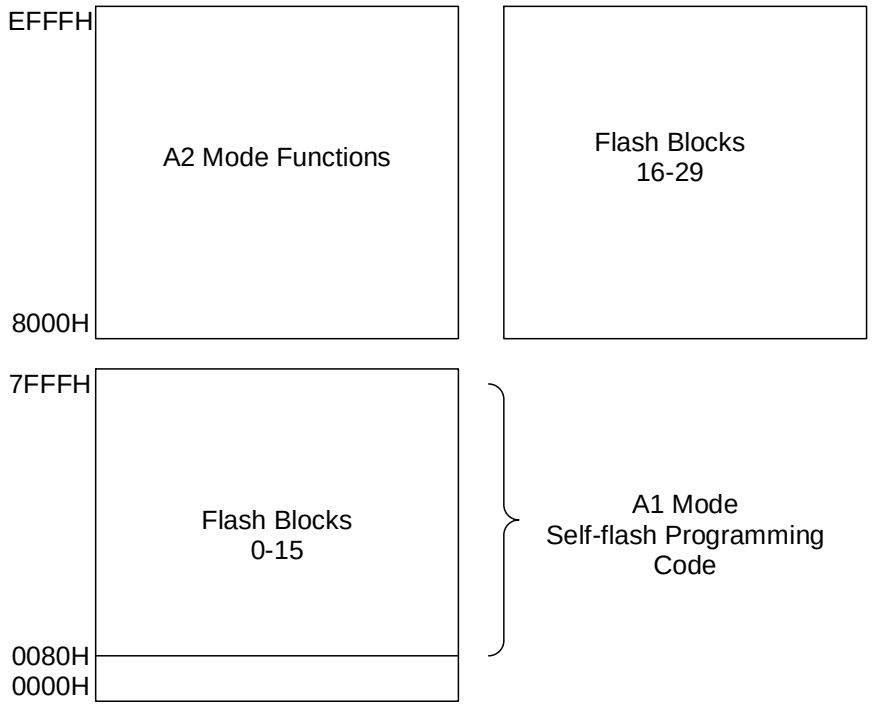
These MCU resources are used:

- 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, etc., 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. This restriction also means that code cannot be executed from flash blocks 16–29 in A1 mode, although the blocks may be erased, programmed, verified, etc.

After using the hidden ROM functions, users should immediately return to normal operation where there is no access to hidden ROM functions.

Figure 5. A1 Mode Memory Map



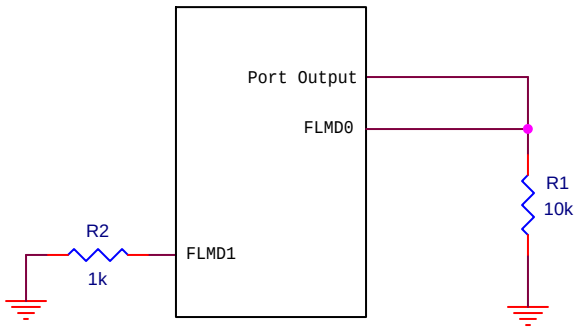
2.5 A2 Mode

In A2 mode, the MCU executes hidden ROM functions and then returns to the calling routine in A1 mode.

2.6 Hardware Requirements

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

Figure 6. Circuit for FLMD0 and FLMD1 Pins



2.7 FLMD0 Pin

Pin FLMD0 normally is low but must be pulled high for flash self-programming. The recommended configuration is shown in Figure 6, 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.

2.8 FLMD1 Pin

Pin FLMD1 should be kept low. If FLMD0, FLMD1 and RESET are all high, then the MCU enters a test mode. Since FLMD0 is set high for flash self-programming, it is advisable to keep the FLMD1 pin low. Although FLMD1 can alternately function 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.

3. Bootloader Configuration and Operation

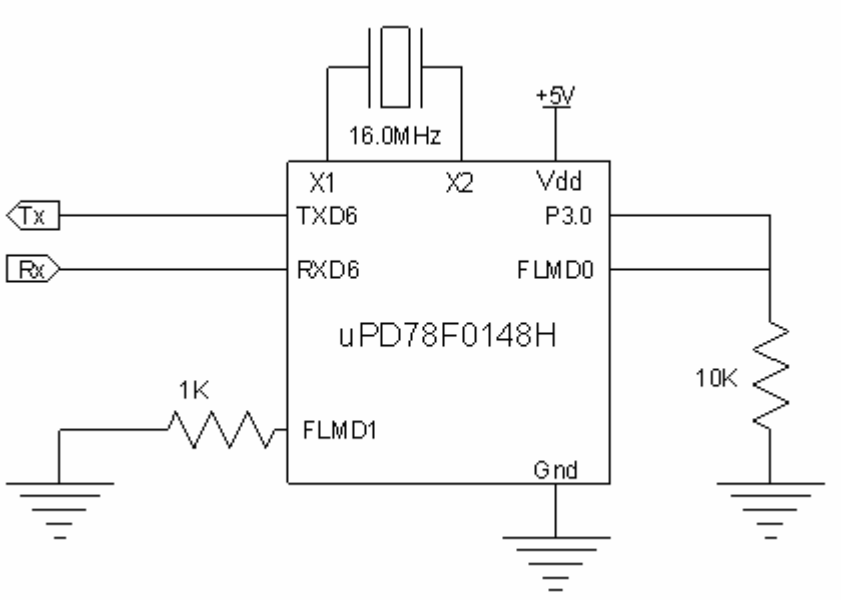
This section describes the bootloader's configuration and operation during the loading of new application code.

3.1 Bootloader Circuit

The bootloader described in this document is configured as shown in Figure 7:

- µPD78F0148H MCU
  - 16 MHz clock
  - FLMD0 pin tied to ground via 10K resistor
  - FLMD1 pin tied to ground via 1K resistor
  - P3.0 port output used to control state of FLMD0 pin
  - UART6 serial interface

Figure 7. Bootloader Circuit



3.2 Bootloader Communication

The UART6 used by the MCU for transmitting and receiving serial data is configured as follows:

- 115200 baud
- 8 data bits
- No parity
- One stop bit
- XON/XOFF flow control

3.3 Bootloader Operation

Figure 8 outlines the **Erase**, **Program**, and **Verify** commands of the application code using XON/XOFF flow control to download the addresses and data.

3.3.1 Boot Prompt for Loading Application

When the MCU boots up, either on power-up or after a reset, it displays the following prompt:

```
NEC Electronics Inc. Bootloader Version X3
Load Y/N?
```

If you enter **Y**, then you will be prompted to download the new application file, which is expected to be in standard Intel hexadecimal 16 code as detailed in Appendix C. Otherwise, if you enter **N**, or if there is no entry for a number of seconds, the MCU proceeds to check for valid application code to execute.

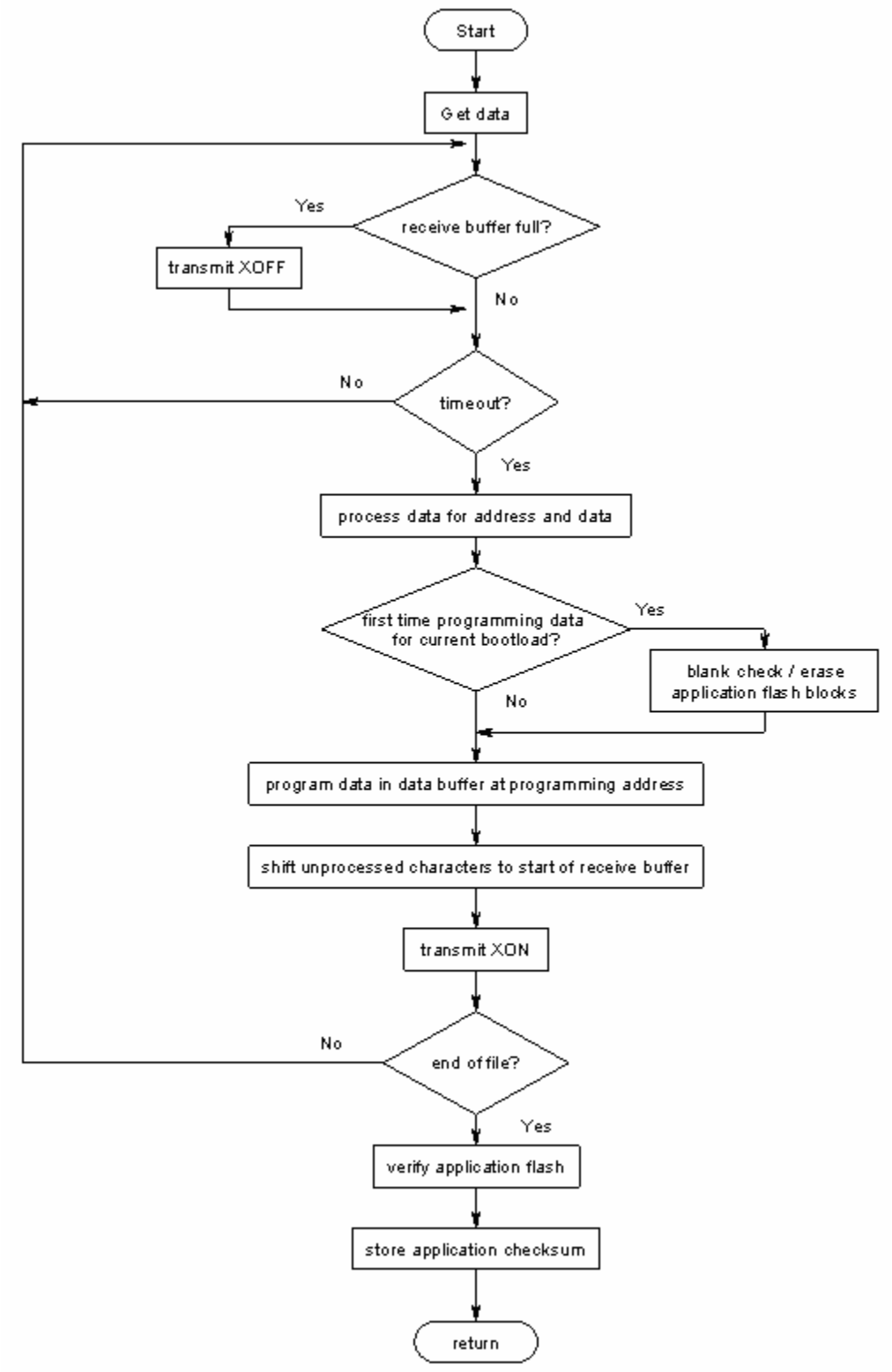
3.3.2 Loading the Application

When you enter **Y**, then the bootloader prompts you to send the file:

```
Send File
```

You then can use the **Send** or **Transfer File** command in your terminal program to send the file.

Figure 8. Bootloader Flow Using XON/XOFF Control



3.3.3 Receiving the Data

The bootloader loads the incoming data into a receive buffer. When the buffer is full, the bootloader transmits the XOFF character to halt the flow of data and continues to process any remaining data until no more characters are received. Then a timeout causes an exit from the receive loop and begins processing of the data. Note that the same timeout mechanism comes into play if characters stop being received before the receive buffer is full.

3.3.4 Processing the Received Data

The bootloader examines the data in the receive buffer and executes the following tasks:

- Checks that the line checksums are correct
- Checks for breaks in line addresses (non-contiguous addresses)
- Sets the programming address
- Extracts the data to be programmed into the data buffer
- Programs the data
- Shifts the unprocessed data to the start of the receive buffer
- Checks for end of file
- Transmits XON before returning to receive loop

The bootloader processes the receive buffer and stores a start address for programming. It then stores data from contiguous addresses into the data buffer. When all the data from the receive buffer has been processed, or if there is a break in the address sequence, then the bootloader programs the data in the data buffer. Afterward, the bootloader shifts the data in the receive buffer, to move any unprocessed data to the start of the buffer, and then transmits the XON character and returns to the receive loop where it left off. The bootloader continues this XON/XOFF receive/programming sequence until it detects the end of the file.

3.3.5 Blank checking and Erasing

After the bootloader has processed the first set of data from the incoming file, it executes a **Blank Check**/**Erase** command before programming the application code into flash memory. Up to this point, if an error in the process causes the bootloader to reset, the application flash will still be valid. The bootloader blank checks the flash memory first and only erases those blocks that are not blank. If block are erased, then the bootloader output the numbers of those blocks in hexadecimal format.

```
Erasing 02
Erasing 03
Erasing 1D
```

3.3.6 Programming

The bootloader programs the data in the data buffer by specifying the address and the number of words to be programmed. Each time the bootloader programs data, it outputs the starting addresses so that you can monitor the progress of bootloading, as follows.

```
Programming at ..
101C
102C
112C
122C
132C
Etc.
```

3.3.7 Verifying

When the bootloader has received and programmed the whole file without errors, then it verifies the flash memory and outputs this message:

```
Verifying..
```

3.3.8 Storing a Valid Application Checksum

When application code is successfully verified, the bootloader stores a 4-byte checksum at the end of flash memory. This checksum is summed over the entire flash application area, from the start of FIRST_FLASH_BLOCK to the end of LAST_FLASH_BLOCK, excluding the four bytes used for storage.

```
#define FIRST_FLASH_BLOCK 2                 // first application block
#define LAST_FLASH_BLOCK 29                 // last application block
```

The four bytes used for storage are the last four locations of the last flash block. If the size of flash memory is greater than 32 KB, then the MCU must be in normal mode before the checksum can be calculated. In A1 mode, the MCU uses addresses greater than 8000H for hidden ROM A2 mode functions. After the checksum has been programmed in normal mode, the block is verified.

```
Checksum 00DC8195 at..
EFFC
Verifying..
```

Note: In this example, the checksum is calculated by summing the data bytes from 1000H to EFFBH. The checksum is then stored as a 4-byte word in the last four locations of block 29 (EFFCH, EFFDH, EFFEH, EFFFH).

3.3.9 Executing the Application

Immediately after successfully loading the new application code, the MCU proceeds to execute it. Whether the code is written to run immediately after a bootload, or after power-up or a reset, the bootloader always checks for a valid application checksum before executing the application code. The bootloader calculates a checksum for the flash application memory, compares it against the stored checksum, and then outputs the information in this form:

```
Stored checksum     = 00DC8195
Calculated checksum = 00DC8195
```

If the values are equal, the MCU proceeds to execute the application code. Otherwise, the program resets and prompts you to download a new application. This valid application check guards against execution of invalid code, which could happen, for example, if an MCU is programmed with the bootloader but has not yet received any application code. Invalid execution also could happen if noise or power loss causes an MCU reset or if one of those conditions occurs in the middle of a download.

3.4 Error Reporting

The following errors may be reported during a bootloading procedure.

3.4.1 FLMD1 Pin High

As shown in the bootloader circuit in Figure 7, the FLMD1 pin should be low. When the MCU enters A1 mode, the FLMD0 pin must be set high. If the FLMD0, FLMD1 and RESET pins are all high, then the MCU enters a factory-set test mode. To guard against this, the bootloader checks the level of the FLMD1 pin before setting FLMDO high when entering A1 mode. If the bootloader finds that the pin is high, then the bootloader outputs this message:

```
FLMD1 high!
```

The bootloader then enters an endless loop and allows the watchdog timeout to force a reset condition.

3.4.2 Flash Self-Programming Errors

If the bootloader encounters any errors when entering A1 mode, or when using any of the flash self-programming functions, the bootloader outputs this type of message:

```
SFP:  Function = 04    Return = 05
```

The bootloader then enters an endless loop and allows the watchdog timeout to force a reset.

Note: Function numbers and return values are detailed in Appendix B.

3.4.3 Missing First Colon

To guard against missing the first characters of a file, the bootloader does not accept any file sent without a colon as the first character. If the colon is missing, the bootloader outputs this message:

```
1st : missing!
```

The bootloader then enters an endless loop and allows the watchdog timeout to force a reset.

3.4.4 Incorrect Line Checksum

As the characters in the receive buffer are being processed, the bootloader examines each line for the correct checksum and, if an incorrect one is found, the bootloader outputs this message:

```
Line checksum!
```

The bootloader then enters an endless loop and allows the watchdog timeout to force a reset.

3.4.5 Boot Area Overwrite

When bootloading a new application and before programming the received data, the bootloader examines each address to make sure that it is outside the boot cluster 0 area of 0000H–0FFFH. If any programming address falls inside this range, the bootloader does not program this address as it would corrupt the bootloader code. Instead, the bootloader outputs this message:

```
Boot Area!
```

The bootloader then enters an endless loop and allows the watchdog timeout to force a reset.

4. Bootswapping Feature

For a blank MCU, the bootloader first must be programmed using an external programmer. Afterward the external programmer can be set aside because the bootloader can load new application code by itself. In many MCUs, you must use an external programmer to update the bootloader code, because the bootloader

code cannot execute and overwrite itself at the same time. The 78K0/Kx1+ MCUs, however, allow you to update bootloader code safely by means of a bootswapping feature when all of the bootloader code is in the 4 KB in Boot Cluster 0 (0000H–0FFFH0. The bootswapping feature makes use of the Boot Cluster 1 area, which is the 4 KB from 1000H–1FFFH. If there is application code in this area, it must be rewritten after the bootswapping procedure is completed. The following description of the bootswapping procedure assumes that there is application code in the Boot Cluster 1 area.

4.1 Bootswapping Procedure

Figure 8 details the bootswapping procedure.

4.1.1 Bootswapping Step 1

The MCU is reset and Bootloader A (Boot Cluster 0) starts executing. When the user is prompted to download a new application, they select Bootloader B for download.

4.1.2 Bootswapping Step 2

Bootloader A determines that new bootloader code examines the first address of the file being sent. If this address is the reset address of 0000H, the Bootloader A knows that the file being sent is new boot code rather than an application. After determining that the file is new boot code, the bootloader must erase boot cluster 1 (blocks 2 and 3).

```
Erasing
02
03
```

4.1.3 Bootswapping Step 3

Although Bootloader B has addressing in the 0000H–0FFFH range (Boot Cluster 0), Bootloader B must be stored in the 1000H–1FFFH range (Boot Cluster 1), which is accomplished by adding an offset of 1000H to all addresses before programming. The program starting address output looks like the following:

```
Programming..
0000->1000
0100->1100
0200->1200
0240->1240
Etc.

Verifying..
```

If there are no errors in the process, Bootloader A prompts you to confirm:

```
Replace the boot code Y/N?
```

If you enter **Y**, then Bootloader A sets the bootswapping flag.

Whether the bootswapping flag is set or not, Bootloader A now allows a watchdog timeout to force a reset.

4.1.4 Bootswapping Step 4

When reset, the MCU determines that the bootswapping flag is set and starts executing the code in Boot Cluster 1 (Bootloader B).

4.1.5 Bootswapping Step 5

Bootloader B determines that the bootswapping flag is set and knows it must copy itself to Boot Cluster 0 after first erasing Boot Cluster 0 (blocks 0 and 1)

```
BootSwapping..
Erasing 00
Erasing 01
```

4.1.6 Bootswapping Step 6

Bootloader B copies itself into Boot Cluster 0 (blocks 0 and 1), clears the bootswapping flag, and allows the watchdog timeout to force a reset

4.1.7 Bootswapping Step 7

After a reset, the MCU determines that the bootswapping flag is clear and then starts executing the code in Boot Cluster 0 (now Bootloader B).

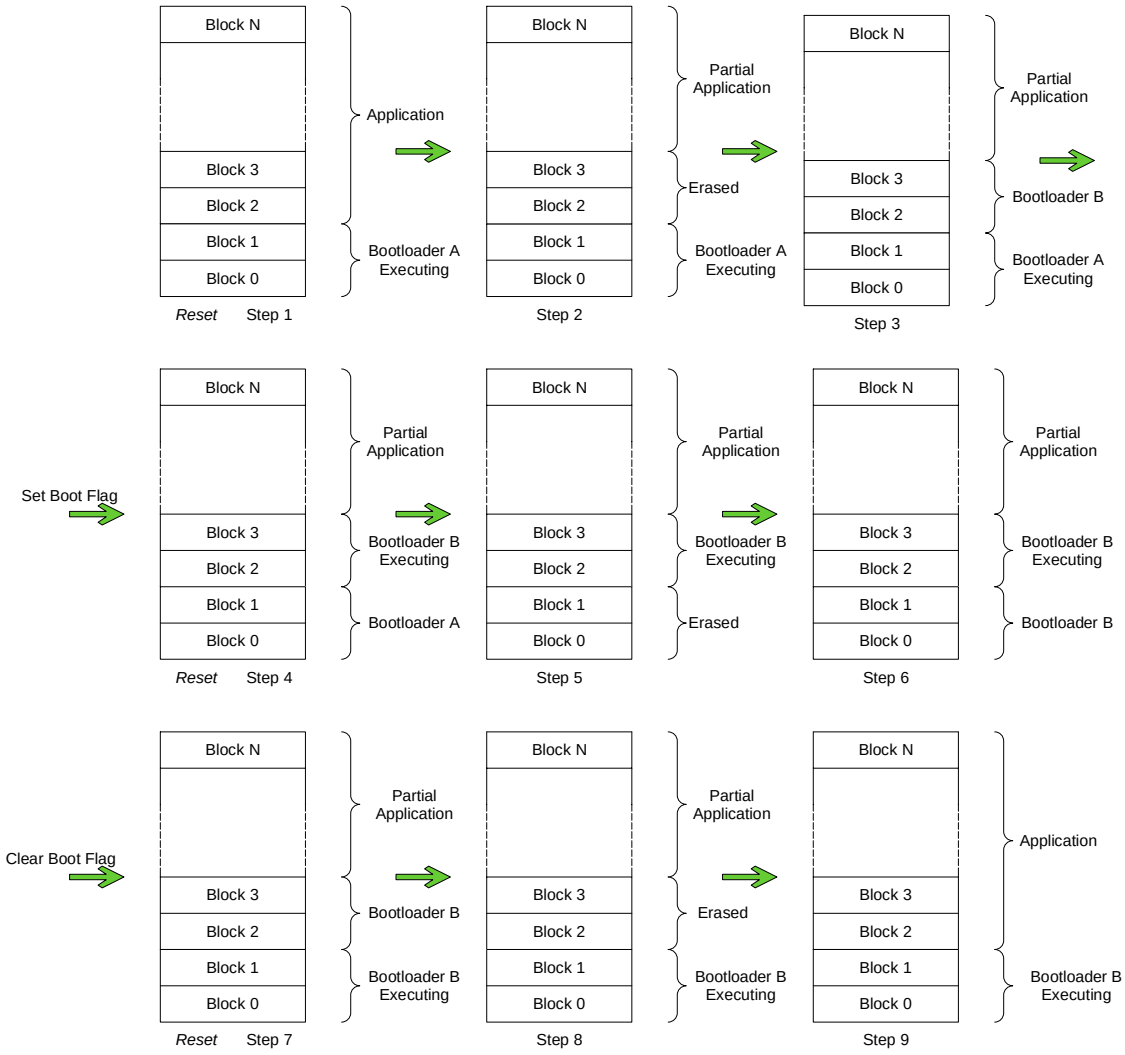
4.1.8 Bootswapping Step 8

After prompting for new application code, Bootloader B now erases Boot Cluster 1 so it can load the missing application code. If Bootloader B needs to load the whole application, it will erase blocks 2 to N (where N is the last block number needed by the application code).

4.1.9 Bootswapping Step 9

Bootloader B loads the partial application code that is missing (or else reloads the whole application).

Figure 9. Bootswapping Procedure



4.2 Interruption of the Bootswapping Process

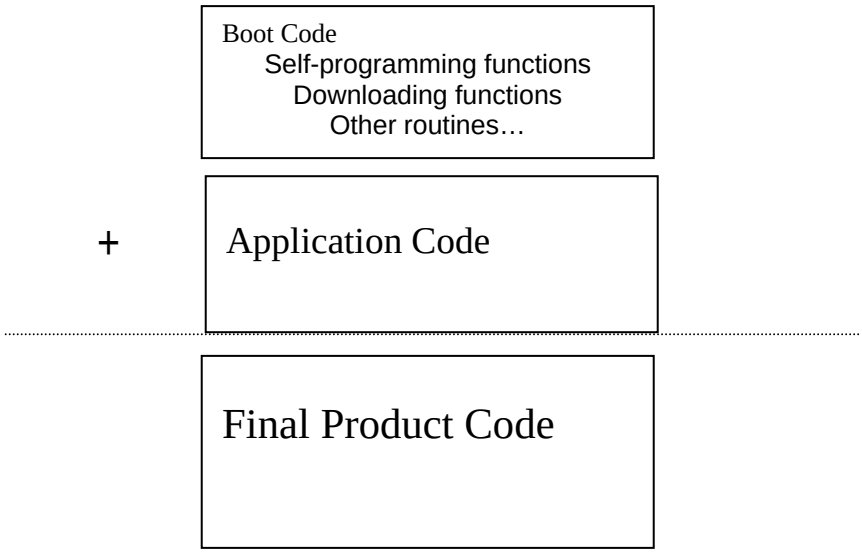
If noise or power loss causes the MCU to reset at any stage in this process, the MCU will still boot up correctly because it retains a full copy of the bootloader code throughout the above process, and the MCU can use the bootswapping flag to determine which boot cluster to execute boot code from.

5. Developing Self-Programming Code Using NEC Electronics Tools

5.1 Code Structure

To take full advantage of the self-programmability features of the 78K0/Kx1+ MCUs, it is recommended to separate your code into two sections; the boot area and the application area. By separating the code in such manner, you can update the application code without disrupting a bootloader program. In addition, the bootloader itself can be updated by using the bootswapping function available in the 78K0/Kx1+ MCU.

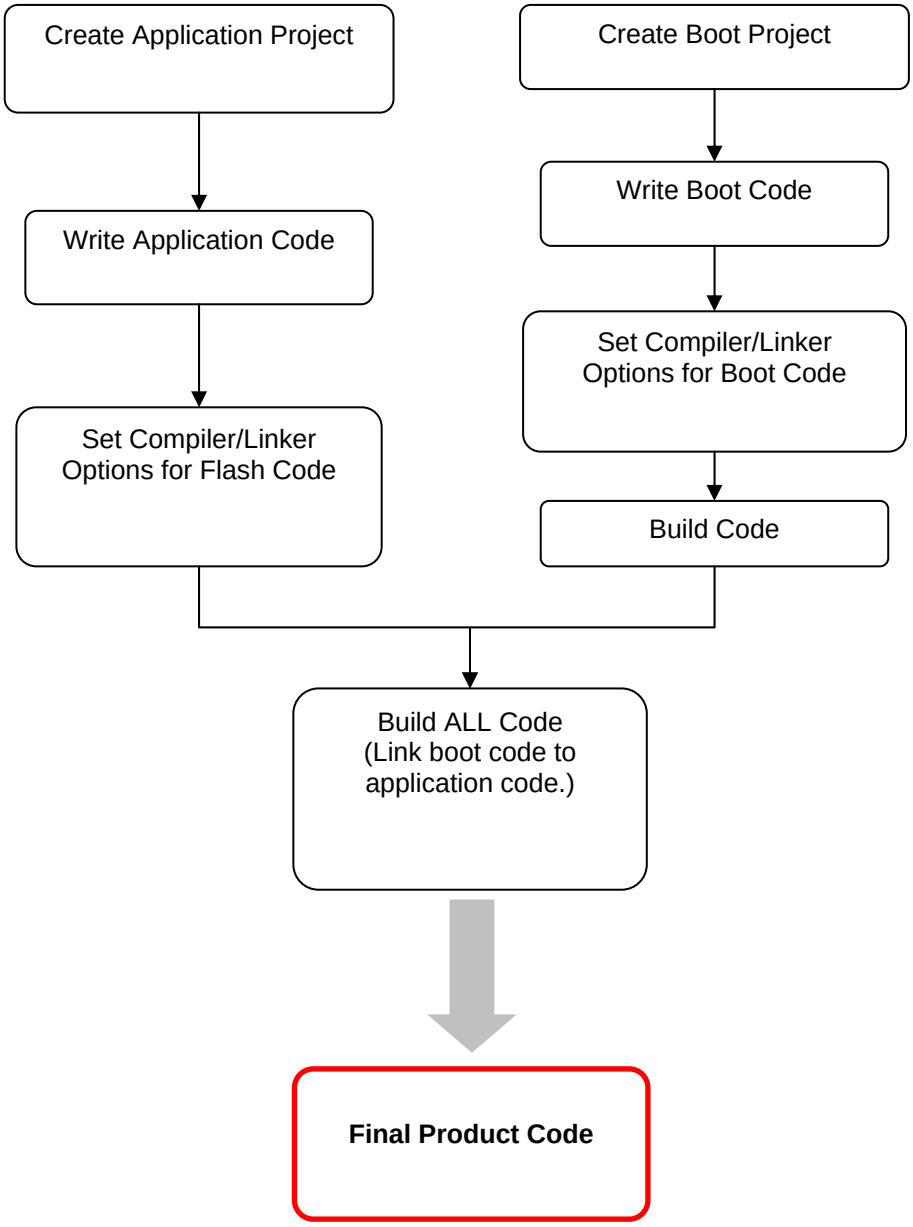
Figure 10. Self-Programming Code Structure



5.2 Development Flow

The typical development flow for generating code with self-programming capabilities using NEC Electronics tools is shown below.

Figure 11. Code Development Flow

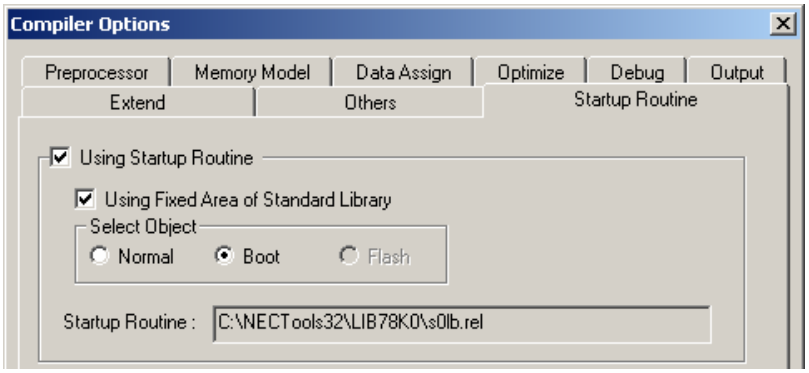


5.3 Tools Setup

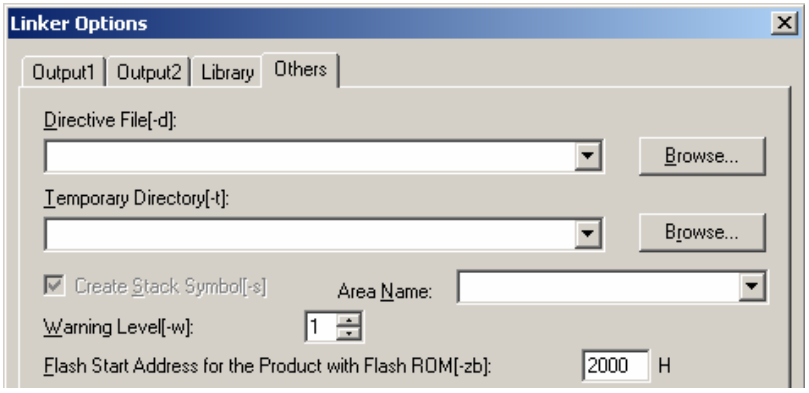
NEC Electronics' tool suites provide everything necessary to develop code with self-programming functionality. Self-programming code generation can be achieved easily using the compiler, linker and object converter options provided through NEC Electronics' PM Plus Integrated Development Environment (IDE) user's interface. Below is a description of the minimum number of steps that you can follow to generate boot and application code with NEC Electronics tools' default options. For details on this procedure, refer to the CC78K0 Language and Operation Manuals.

5.3.1 Procedure to Generate Boot Code

1. Main() function must be named **boot_main()**
2. Specify **C Startup Routine for Boot** in C compiler options.



3. Specify the starting address of application code (default value should be 2000H).



Notes:

- If application code starts at an address other then 2000H, you must rebuild the NEC Electronics run-time libraries. Read the steps described in the *CC78K0 Language User's Manual* for more detail.
- The bootloader example developed by NEC Electronics uses 1000H.

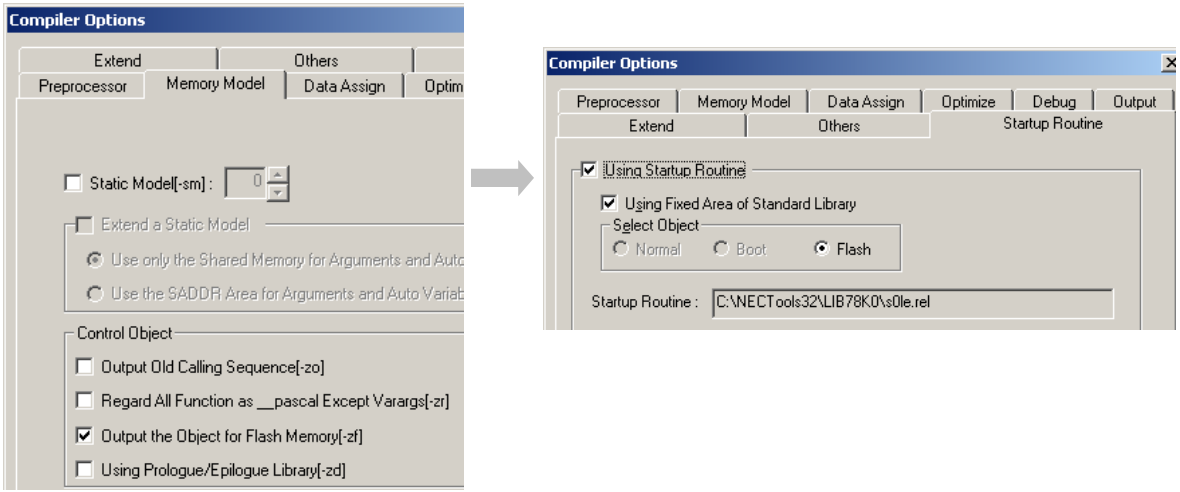
5.3.2 Procedure to Generate Application (Flash) Code

1. Specify in code the starting address of the application code. At the very top of the C language source file, write the following:

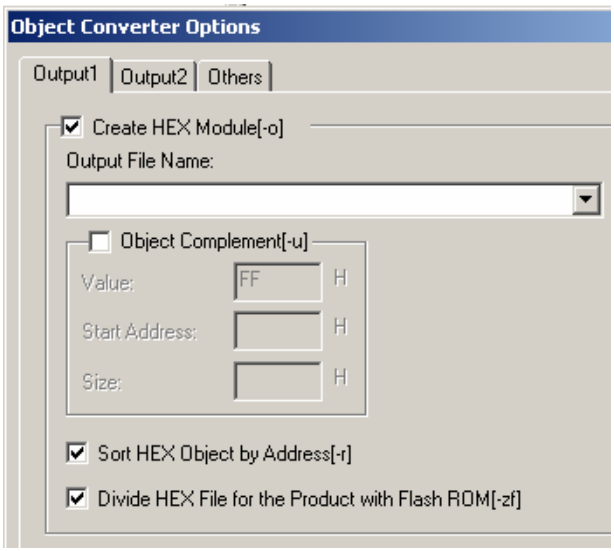
```
#pragma ext_table 0x2000          //using address 2000H as example
```

Note: This directive defines the first address of the application code, which is used by the C startup routines and interrupt functions.

2. Specify to output object code for flash memory (application code). This setting automatically selects the correct C startup routine for flash code.



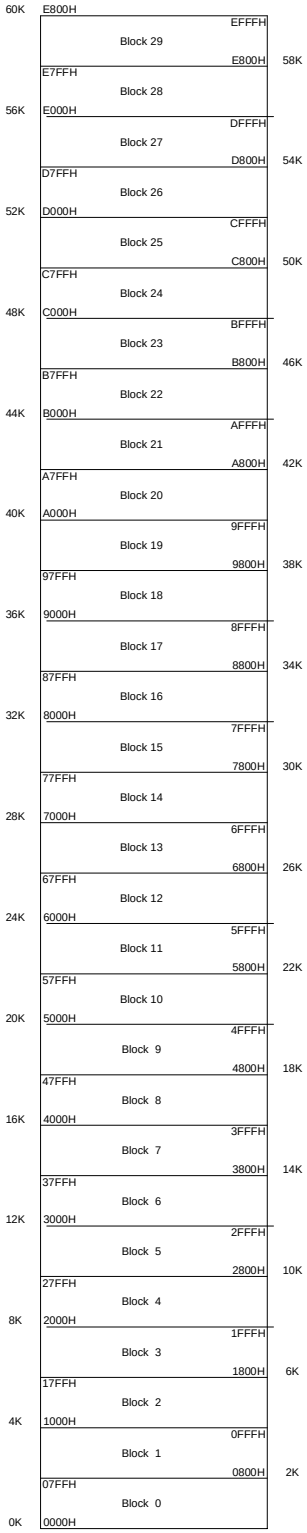
3. In the **Object Converter** options, specify to generate two separate HEX files, one for boot code and one for application (flash) code.



Notes:
The file extensions are as follows:

- *.HXB (hex file for boot code)
- *.HXF (hex file for flash/application code)
- File *.HXB should match file *.HEX generated from boot code build.

Appendix A. µPD78F0148H Flash Memory Block Addresses



Appendix B.
Flash Self-Programming Function Numbers and Return Values

Table 1. Flash Self-Programming A2 Mode Functions Used with Bootloader

Function	Function Number	Return Value
Initialization	00H	00H: Normal completion 05H: Parameter error
Block Erase	03H	00H: Normal completion 05H: Parameter error 1AH: Erasing error
Word Write	04H	00H: Normal completion 05H: Parameter error 18H: FLMD0 error 1CH: Write error
Block Verify	06H	00H: Normal completion 05H: Parameter error 1BH: Internal verification error
Block Blank Check	08H	00H: Normal completion 05H: Parameter error 1BH: Blank check error
Get Information	09H	00H: Normal completion 05H: Parameter error
Set Information	0AH	00H: Normal completion 05H: Parameter error 18H: FLMD0 error 1BH: Internal verification error 1CH: Write error
Mode Check	0EH	00H: Normal completion 01H: Error

Table 2. Other Flash self-programming Functions Used with Bootloader

Function	Function Number	Return Value
Enter A1 Mode	A1H	00H: Normal completion 01H: Protection error
Return to Normal Mode	5FH	00H: Normal completion 01H: Protection crror

Appendix C. Intel Hexadecimal 16 Format

This is a text file containing the addresses and data of the code. Each line is one record and contains up to 16 bytes of data.

Table 3. Intel Hex 16 Code Format

Character Position	No. of Characters	Name	Description
1	1	Record Marker	Colon character ':' (3A hex)
2–3	2	Record Count	2-character hex byte giving number of data bytes
4–7	4	Address	4 character hex address
8–9	2	Record Type	00 = Data 01 = End of file 02 = Extended address
10–?	2-32	Data Bytes	2-character hex data bytes (1 to 16)
Last 2	2	Checksum	2's complement of sum of hex bytes in record (excluding record marker and vhecksum itself)

Example of record with five data bytes at address 2400 hex:

:052400000102030405C8