APPLICATION NOTE

H8/300L SLP Series

SLP User Mode Programming (UserMP)

Introduction

This application note provides the complete solution for user mode flash memory programming on SLP microcomputer. The document comes with the source codes for:

1. User mode programming kernel
2. User mode demo program
3. Flash programming GUI (TCL/TK based software)

Target Device

H8/38024F

Contents

1. Overview 2
2. GUI 4
3. UI (User Interface) 10
4. Kernel 11
5. Application 12
6. Communication Protocol 13
7. MCU Coding Implementation 14
8. Overall Operation and Observations 23
9. Code Listing 25
10. Serial Communication Debugging Technique 60
11. References 61

1. Overview

The MCU with on-chip flash memory has two modes of operations: the boot mode and the user mode.

In **the boot mode**, the MCU expects to communicate with the external world through its serial port. This is to 'program' the MCU on-chip flash memory as there is no program in the MCU at this initial startup state. This boot mode flash programming has been detailed in the application note "In-circuit boot mode programming". (In this mode, the user is not required to write any code, as a boot mode kernel is residing in the MCU)

Once the MCU has been programmed, it can power up in the **user mode** for the program execution. In the user mode, the flash memory can be (re)programmed. However, user will have to prepare the user kernel, host interfacing program and also the host control software (which can be a PC or another embedded system)

1.1 Boot Mode Programming

The boot mode provides an automated mechanism to program a blank device in-circuit, or to reprogram a device with an automatic chip-erase prior to programming. When the boot mode is entered from chip reset, the boot program in the LSI (originally incorporated in the chip) is started to provide the following services:

1. Serial port auto baud rate detection with external host
2. Download of a user supplied boot kernel into RAM via the serial port
3. Erase program in the boot program is executed to erase all the contents of the flash memory
4. Execution of the downloaded boot kernel

The example below shows the signals required for entering and exiting the boot mode for single rail programming devices (check the Hardware Manual for device specifics):

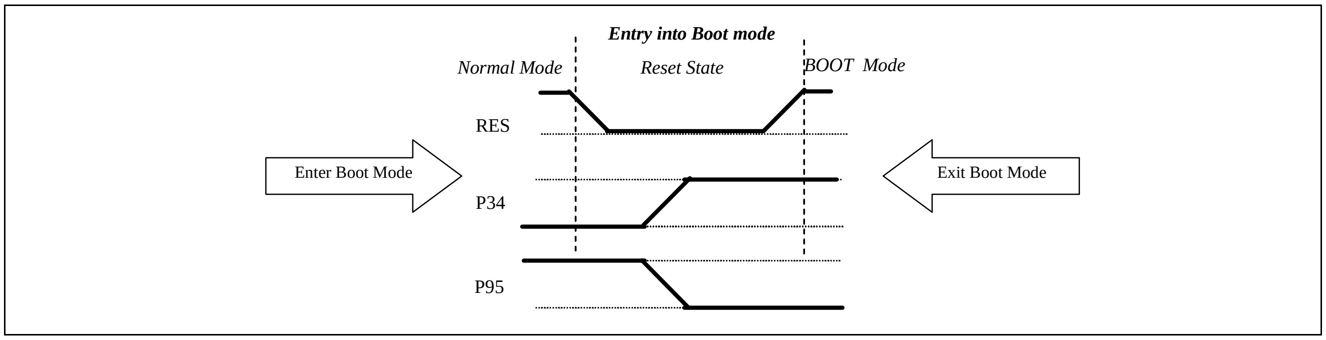


Figure 1 Boot Mode Entry Timing Diagram

At the point of execution of the boot kernel, the entire chip is erased, and ready for programming. The boot kernel itself can perform any function (as this is a user supplied application), however it should include a programming function, as the chip is now blank! The boot kernel may continue to use the serial port for data download, or can use any other peripheral features of the chip to acquire the required data (e.g. parallel interface, CAN bus etc.). Upon completion of the programming operations, the required mode pins should be reset to normal execution values and the chip reset.

1.2 User Mode Programming

User mode flash programming allows flexibility in version upgrade, data update etc., which will only change part of the total flash memory without resetting the MCU. Since it is user determined, the data media can come from the serial port or any other communication channel.

To perform programming in the user mode, the following components are essential:

- **Host controller (GUI)** is another system that is communicating with the MCU. It provides the stream of data to be 'burn' into the flash memory of the MCU. In this application note, a PC is used as a host controller. The software used for this GUI is written based on the TCL/TK scripts.
- **User mode host–interfacing routine (UI)** work as the interfacing software to host, which determines the communication channel and data transfer protocol.
- **User mode flash kernel (KERNEL)** is the main controller of the flash reprogramming. It contains the process (0.35-µm flash memory programming algorithm) detail of erasing and programming.
- **Application software (APPLICATION)** is refer to the user target application program that executing the specific embedded system task.

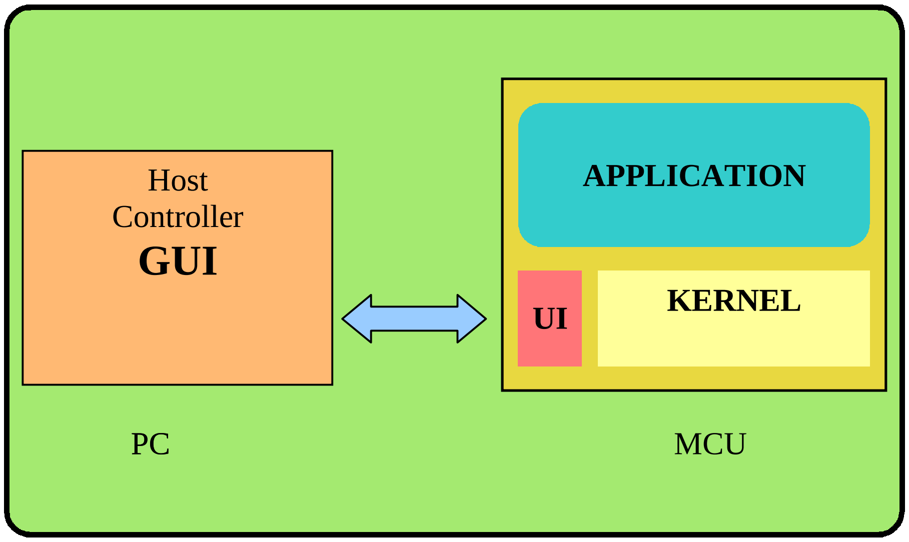


Figure 2 The General View

2. GUI

In both modes, the GUI will:

1. Decode S-record output file into binary format
2. Establish communication with the UI routine located in the MCU
3. Download machine code into MCU via serial port

Two type of flashing are provided in this GUI:

- Boot mode flashing
- User mode flashing

2.1 GUI Overview

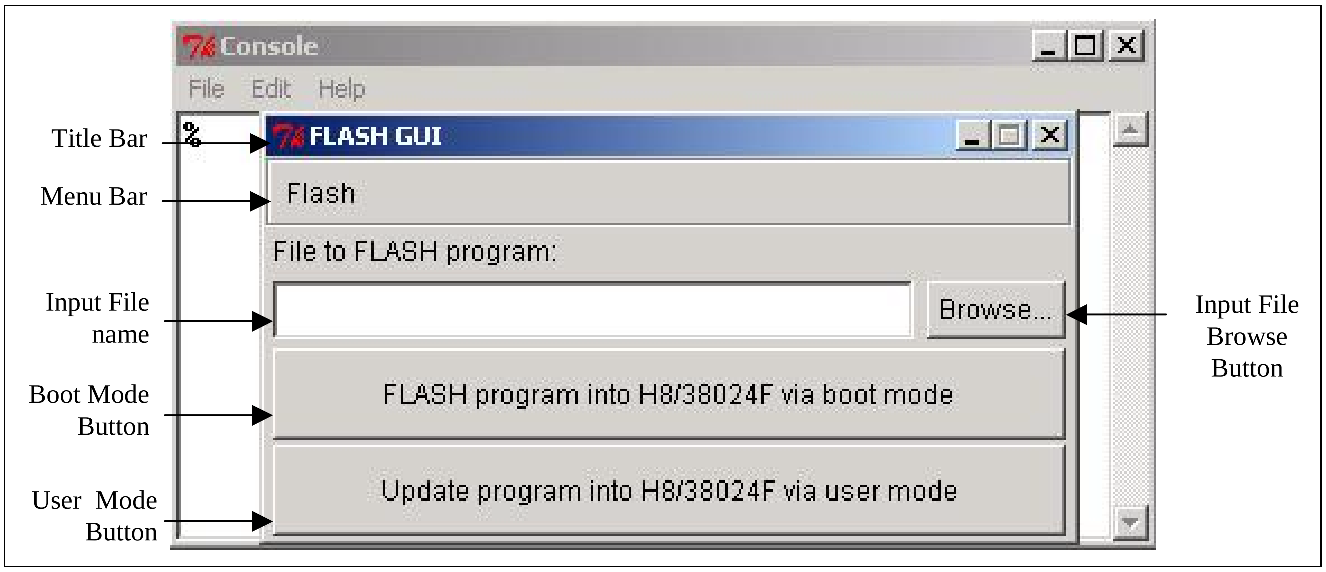


Figure 3 Flash GUI dialog box

Menu Bar: Flash

The user can click on "Quit" in the Flash menu bar to exit Flash GUI.

Input File Name

Flash GUI allow user to select S-Record file to be downloaded into the flash memory.

Input File Browse Button

This command will launch a standard windows open file dialog. The user can only select one S-Record file at each time.

Boot Mode Button

This command is used to download the current input S-Record file. A Flash programming operation writes the data from the selected S-Record file to target Flash memory. This operation is carried out in the boot mode, so the user has to take note that the target device must reset in order to enter the boot mode.

User Mode Button

This command is used to update the target device with current input S-Record file without reset in the boot mode. Please note that, the user must not overwrite or erase interfacing software (located in Flash ROM) during the software update operation. If the user needs to overwrite the whole Flash memory, it's recommended to place the interfacing software in the RAM rather than ROM.

2.2 GUI Scripting Languages

2.2.1 Tcl/Tk Overview

Tcl – Tool Command Language ("tickle") is a simple interpretative programming language.

Some key features of Tcl are summarized as follows:

1. Tcl is a high-level scripting language.
   Users with experience in high-level programming languages should find Tcl similar to the other languages.
2. Tcl is an interpreter
   Code can be executed directly, without compiling and linking.
3. Tcl is extensible
   Users can add their own commands to extend the Tcl language.
4. Tcl is embeddable in applications
   The Tcl interpreter was designed from the start to be embedded in a variety of applications. It is easy to incorporate Tcl into an application, and the Tcl interpreter melds naturally with the application, almost as if the Tcl language was designed exclusively for that particular application.
5. Tcl runs on many platforms
   Supported on Windows, UNIX, and Macintosh platforms, but minor changes have to be made.
6. Tcl is free
   The source for Tcl can be found in internet and can be freely used even for commercial applications.

Tk - Tool kit is a graphical user interface tool for window programming, which works together with Tcl scripting language. It is designed for the X window system, although ports to other window systems are expected to appear eventually. Tk shares many concepts with other windowing toolkits, but the user does not need to know much about the graphical user interfaces to get started with Tk.

Tk provides a set of Tcl commands that create and manipulate *widgets*. A widget is a window in a graphical user interface that has a particular appearance and behavior. The term *widget* and *window* are often used interchangeably. Widget types include buttons, scrollbars, menus, and text windows.

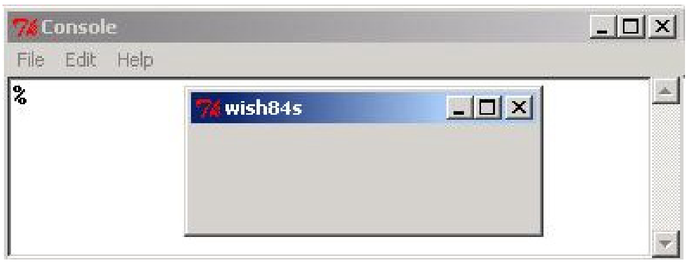
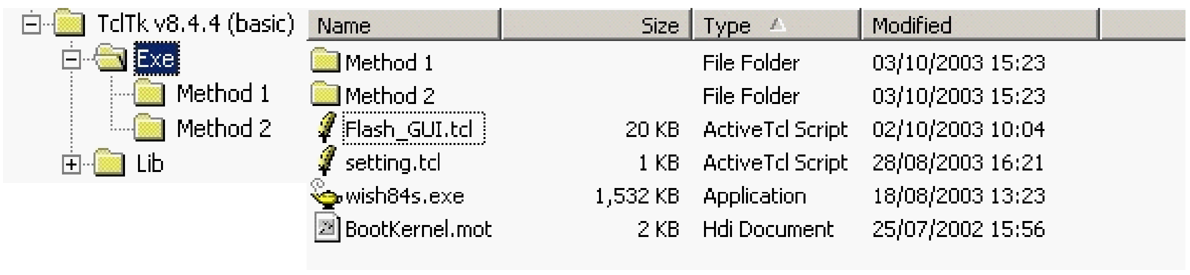


Figure 4 Tcl/Tk scripting interpretive program

2.2.2 TCL/TK LICENSE TERMS

2.2.3 Tcl/Tk Scripting Interpretive Program Installation



Name	Size	Type	Modified
Method 1		File Folder	03/10/2003 15:23
Method 2		File Folder	03/10/2003 15:23
Flash_GUI.tcl	20 KB	ActiveTcl Script	02/10/2003 10:04
setting.tcl	1 KB	ActiveTcl Script	28/08/2003 16:21
wish84s.exe	1,532 KB	Application	18/08/2003 13:23
BootKernel.mot	2 KB	Hdi Document	25/07/2002 15:56

Figure 5 Inside TclTk v8.4.4 (Basic) Folder

2.2.4 Tcl/Tk Scripting Interpretive Program Execution

Double-click “wish84s.exe” to run Tcl/Tk scripting interpretive program.

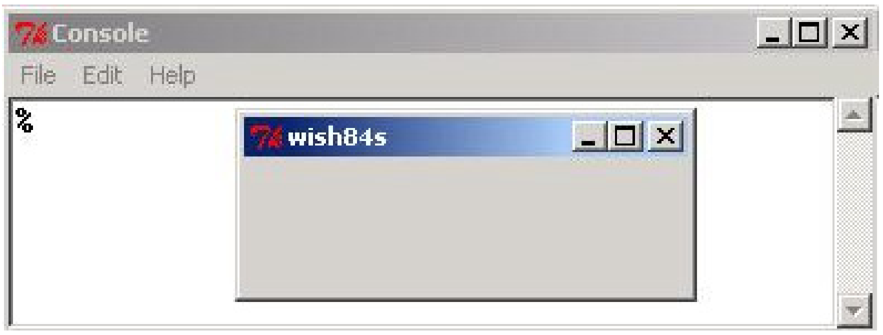
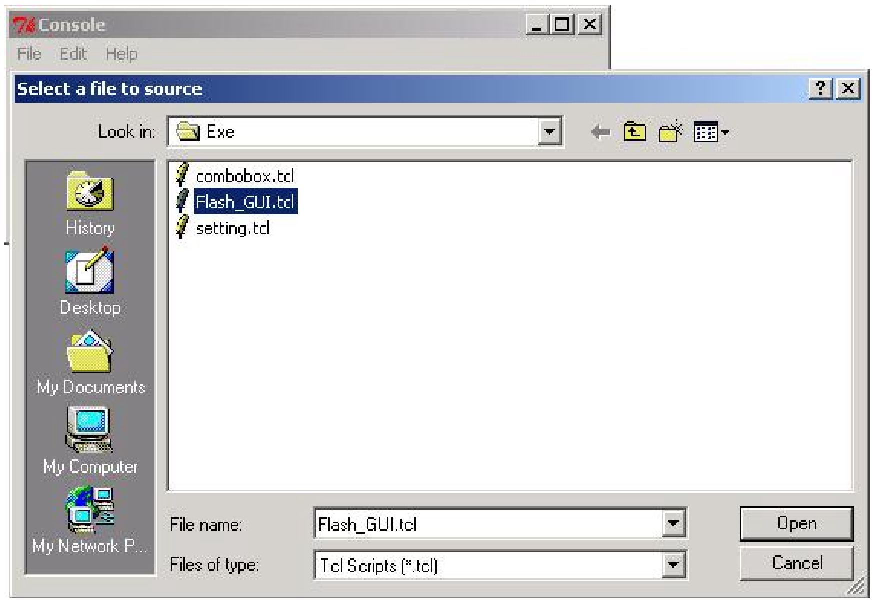


Figure 6 Tcl/Tk Console

Click [File → Sources… → select “Flash_GUI.tcl” → click Open]



File name: Flash_GUI.tcl
Files of type: Tcl Scripts (*.tcl)

Figure 7 Open Flash_GUI.tcl File

2.3 GUI Component

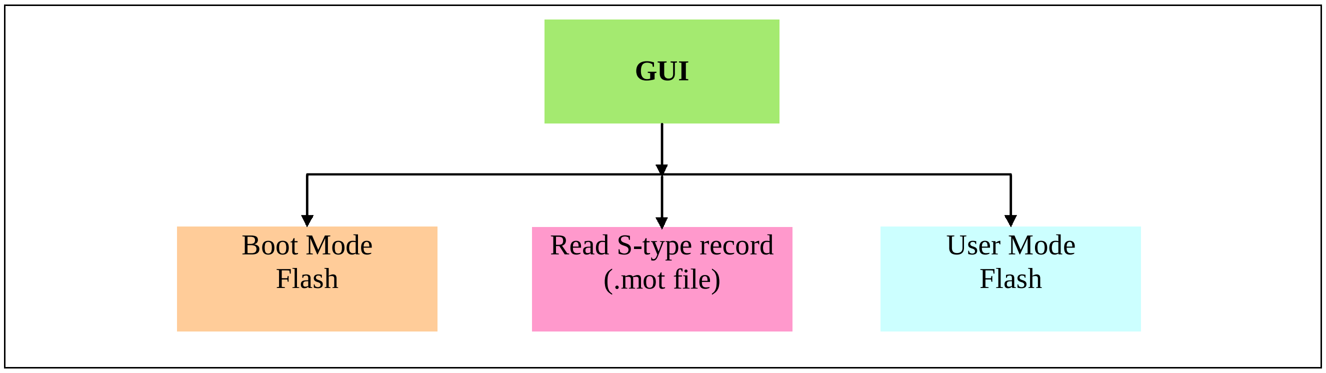


Figure 8 GUI Overview

There are three basic software modules:

1. Read S-type record
   A. Convert S-type record format (.mot) to absolute binary format (.bin)
   B. Break down the binary format data into a block of 128 bytes
   C. Check for empty block information (empty block contains 128 bytes of 0xFF data)

2. Boot mode flash
   — A. Read boot mode flash kernel file
   — Establish boot mode connection with MCU via PC serial port
   — Read the user target program file
   — Download user target program into MCU flash memory
3. User mode flash
   A. Send a write command (character ‘U’) to MCU
   B. Read user target program file
   C. Download the user target program into MCU on-chip flash memory

3. UI (User Interface)

The UI refer to the interfacing routine which determines the MCU communication channel and data transfer protocol.

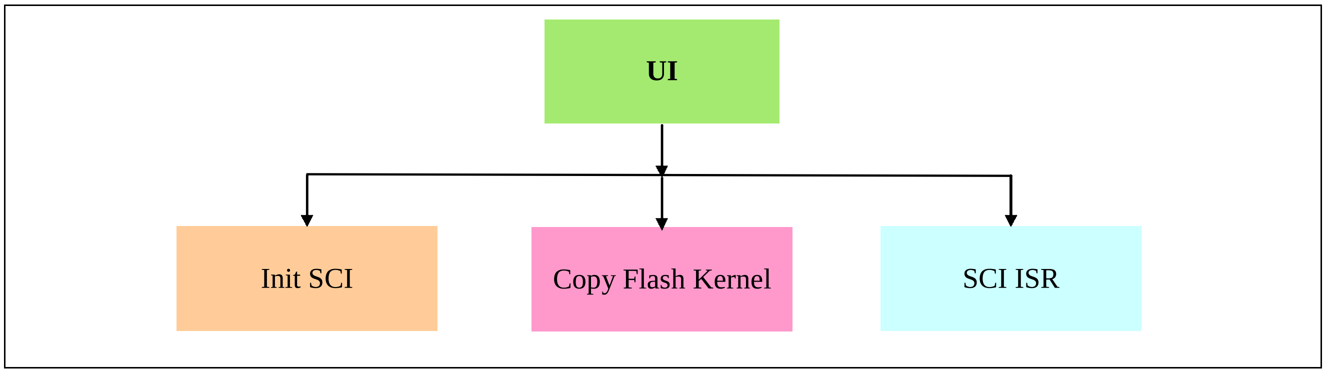


Figure 9 UI Overview

UI Component:

There are three main modules:

1. Init SCI
   — Initialize the on-chip serial communication interface module with receive interrupt enable
   — Set the SCI baud rate to 38400 bps

2. Copy the flash kernel
   — Copy the flash kernel from ROM to RAM
   Note: Flash programming and erasing kernel must be executed in the RAM area

3. SCI ISR
   A. Interrupt service routine for the SCI receive interrupt request
   B. Receive a write command (character ‘U’) from PC
   C. Perform copy flash kernel from ROM to RAM
   D. Get the start address and 128 bytes block data from PC
   E. Call the flash erase routine if an erase block start address detected
   F. Call the flash programming routine (return character ‘a’ if operation passed and ‘n’ if operation failed).
   G. Repeat step (d) until end of flash address detected (0x8000)
   H. Check for data valid flag validation then jump to program reset entry point [PowerON_Reset()]

4. Kernel

The kernel is the flash memory programming routines for H8/38024F microcontroller.

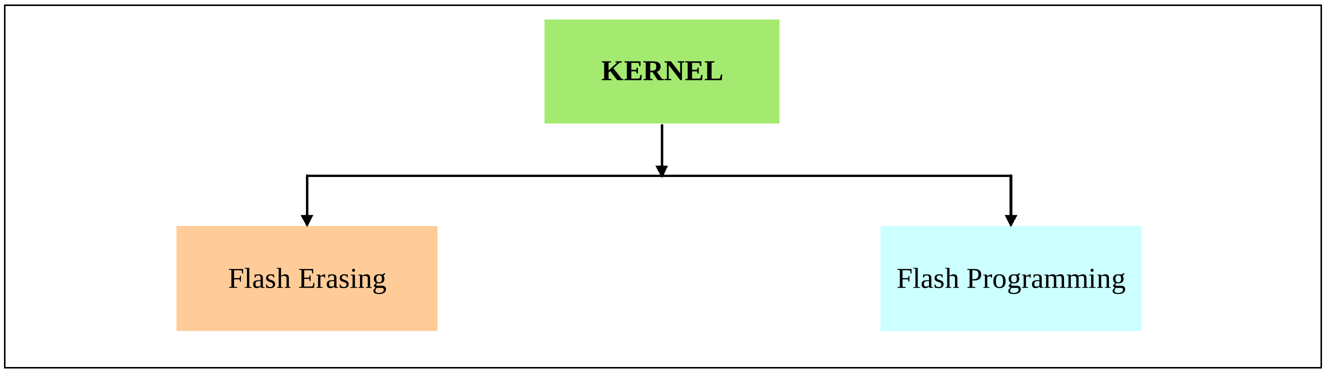


Figure 10 Kernel Overview

Kernel Component'

1. Flash Erasing
   - A. Flash erasing is performed in block units (e.g. Erase Blocks 0, 1, 2, 3 and 4)
   - B. The flash memory is erased in the following process:
     - The flash block is erased
     - The memory is placed into erase-verify mode
     - Flash contents is read back
     - Compared with the erase value of all '1'
   - — If any of the bits in the block are not read back as '1' then another attempt is made to erase the block. This process is repeated until either flash memory block is successfully erased or the maximum number of erase attempts is reached.

2. Flash Programming
   - A. Flash erasing must be performed before flash programming
   - B. The flash memory programming must in units of 128 bytes and starting on a 128-byte boundary (e.g. 0x0000, 0x0080, 0x0100,… , 0x7F00, 0x7F80)
   - C. The 128-byte flash line can be programmed by calling the function 'prog_flash_line_128' in kernel.c file
   - D. The first parameter passed to this function is the start address of the flash memory to be programmed, which must be on the 128-byte boundary.
   - E. The second parameter is a pointer to the data to be programmed.

5. Application

The applicaiton module refers to the user targeted application. This application note consist of a few simple application programs that control two LEDs which are connected to port 9 of H8/38024F MCU (in the SLP 38024F CPU board).

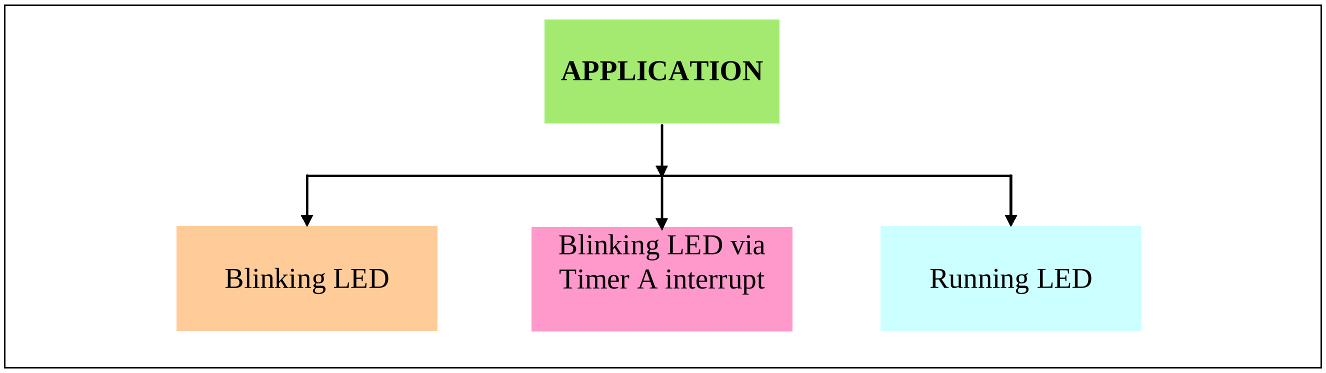


Figure 11 Kernel Overview

Application Component:

Port 9 of the H8/38024F is used as it is a large current port that can drive LED directly without any LED driver.

1. Blinking LED
   A. Two LEDs are connected to Port 9 pin 2 and 3
   B. Application program will toggle port 9 pin 2 and 3 with fixed delay while MCU is running
2. Blinking LED via Timer A interrupt
   A. Two LEDs are connected to Port 9 pin 2 and 3
   B. Timer A overflow interrupt will toggle port 9 pins 2 and 3
3. Running LED
   A. Two LEDs are connected to port 9 pins 2 and 3
   B. Application program will toggle port 9 pins 2 and 3 alternately with fixed delay while MCU is running

6. Communication Protocol

The figure shows the communication protocol between GUI (PC) and UI (MCU) in user mode programming. The boot mode programming is detailed in hardware manual.

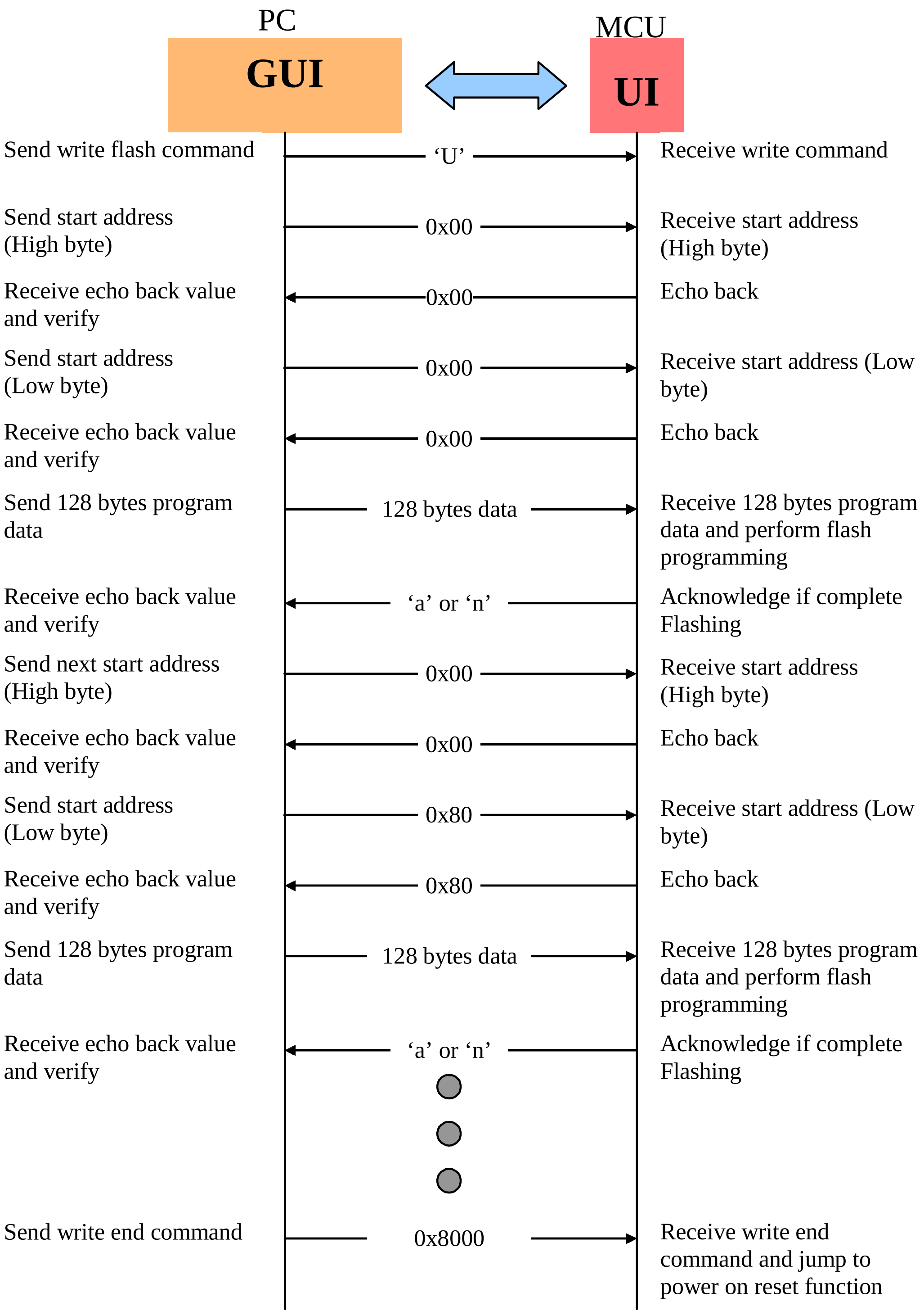

PC
GUI
MCU
UI
Send write flash command
'U'
Receive write command
Send start address (High byte)
0x00
Receive start address (High byte)
Receive echo back value and verify
0x00
Echo back
Send start address (Low byte)
0x00
Receive start address (Low byte)
Receive echo back value and verify
0x00
Echo back
Send 128 bytes program data
128 bytes data
Receive 128 bytes program data and perform flash programming
Receive echo back value and verify
'a' or 'n'
Acknowledge if complete Flashing
Send next start address (High byte)
0x00
Receive start address (High byte)
Receive echo back value and verify
0x00
Echo back
Send start address (Low byte)
0x80
Receive start address (Low byte)
Receive echo back value and verify
0x80
Echo back
Send 128 bytes program data
128 bytes data
Receive 128 bytes program data and perform flash programming
Receive echo back value and verify
'a' or 'n'
Acknowledge if complete Flashing
Send write end command
0x8000
Receive write end command and jump to power on reset function

Figure 12 Communication Protocol Transition Diagram

7. MCU Coding Implementation

A program must reside in the MCU during the user mode execution & programming. This program will contain three main parts:

1. Flash kernel (KERNEL)
2. Host interface program (UI)
3. User application program (APPLICATION)

In order to maintain programmability, the flashing kernel & host interface program must remain in the MCU after any flashing procedure. The main objectives of any flashing procedure are to

1. Update new data, or
2. Upgrade to a new version of user application program

There are two possibilities of works:

1. All blocks
   → The whole MCU flash is erased and a whole new application code (with kernel & host interface program) will be programmed. However this is not a usual programming practice as this is equivalent to Boot Mode programming.
2. Partial block
   → Part of the MCU flash is erased and new code or data is updated.
   → The generation of new data is simple, but user has to pay special attention when generating the new code

The following will elaborate the partial block user programming:

1. Data update
2. Code update
   → Method 1
   → Method 2

7.1 Data Update

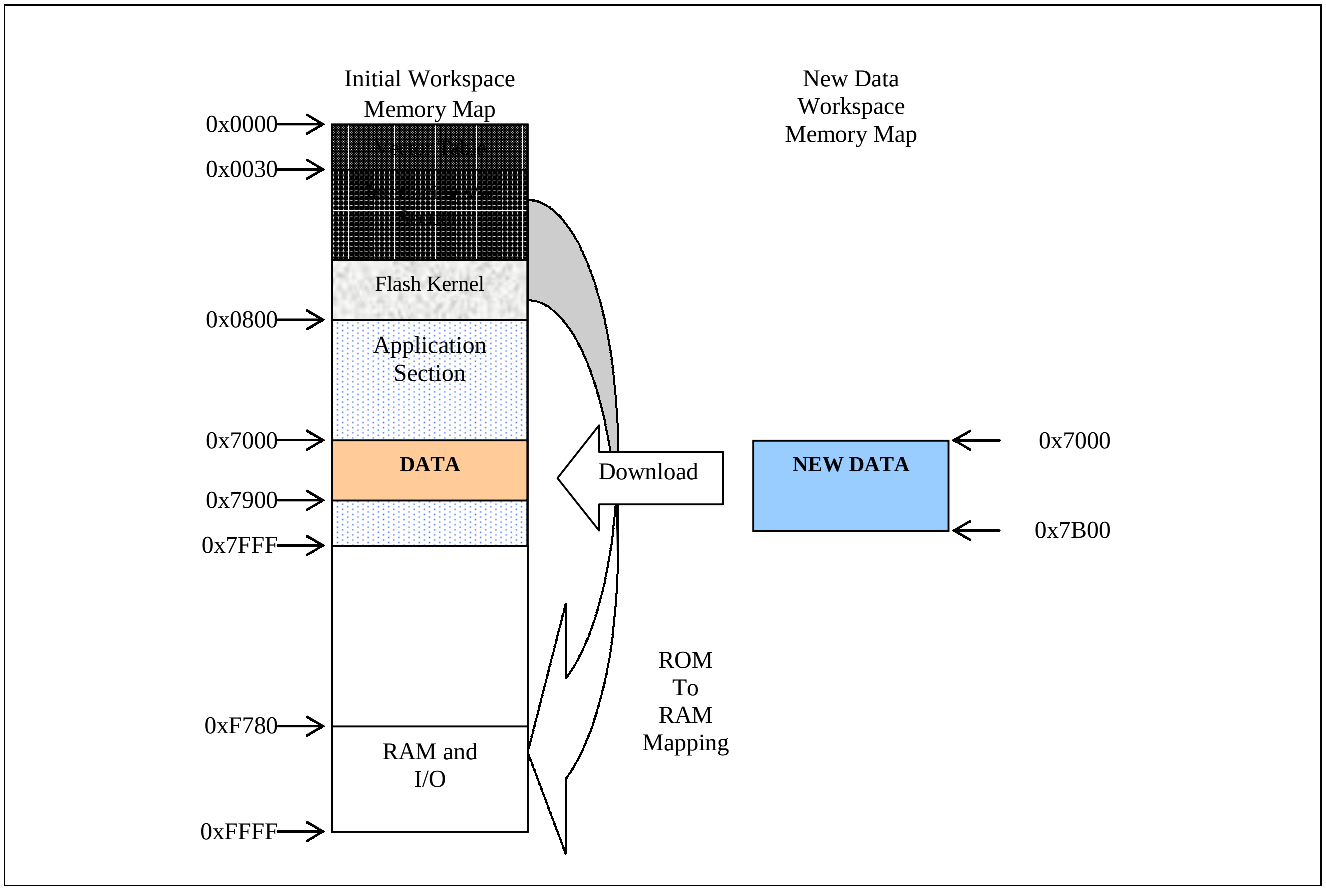
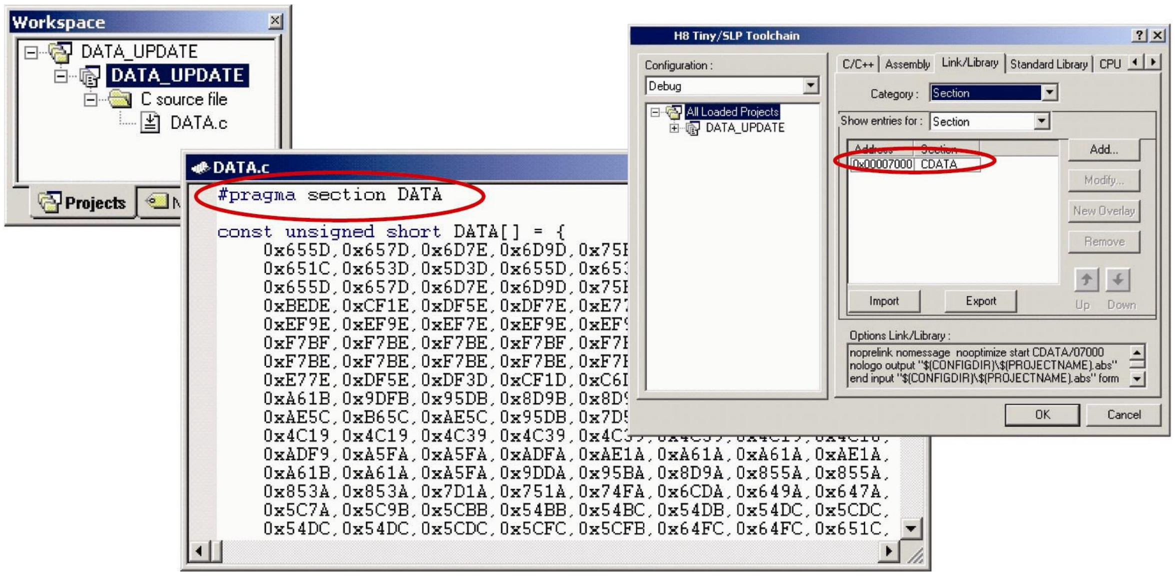


Figure 13 Memory Map for Data Update

Procedures:

1. Create a empty workspace for C or assembly language
2. Declare data (Static constant… or DATA …)
3. Declare the section and define the address.
4. Compile and generate the S record file
5. Alternatively generate the S record file via ‘Save as’ in the emulator/simulator HEW.



```
#pragma section DATA

const unsigned short DATA[] = {
      0x655D,0x657D,0x6D7E,0x6D9D,0x75
      0x651C,0x653D,0x5D3D,0x655D,0x65
      0x655D,0x657D,0x6D7E,0x6D9D,0x75
      0xBEDE,0xCF1E,0xDF5E,0xDF7E,0xE7
      0xEF9E,0xEF9E,0xEF7E,0xEF9E,0xEF
      0xF7BF,0xF7BE,0xF7BE,0xF7BF,0xF7
      0xF7BE,0xF7BE,0xF7BE,0xF7BE,0xF7
      0xE77E,0xDF5E,0xDF3D,0xCF1D,0xC6
      0xA61B,0x9DFB,0x95DB,0x8D9B,0x8D
      0xAE5C,0xB65C,0xAE5C,0x95DB,0x7D
      0x4C19,0x4C19,0x4C39,0x4C39,0x4C
      0xADF9,0xA5FA,0xA5FA,0xADFA,0xAE1A,0xA61A,0xA61A,0xAE1A,
      0xA61B,0xA61A,0xA5FA,0x9DDA,0x95BA,0x8D9A,0x855A,0x855A,
      0x853A,0x853A,0x7D1A,0x751A,0x74FA,0x6CDA,0x649A,0x647A,
      0x5C7A,0x5C9B,0x5CBB,0x54BB,0x54BC,0x54DB,0x54DC,0x5CDC,
      0x54DC,0x54DC,0x5CDC,0x5CFC,0x5CFB,0x64FC,0x64FC,0x651C,
```

Configuration : Debug

Category : Section

Show entries for : Section

Address	Section
0x00007000	CDATA

Options Link/Library :

```
noprelink nomessage  nooptimize start CDATA/07000
nologo output "$(CONFIGDIR)\$(PROJECTNAME).abs"
end input "$(CONFIGDIR)\$(PROJECTNAME).abs" form
```

Figure 14   New Data Workspace Generation

7.2 Code Upgrade

In order to co-exist with the “initial workspace”, the generated code in the “new workspace” has to consider several factors.

1. Initialized variables
2. Stack
3. Constant
4. Entry point to the new workspace
5. Entry point to the initial workspace

7.2.1 Method 1 [M1]

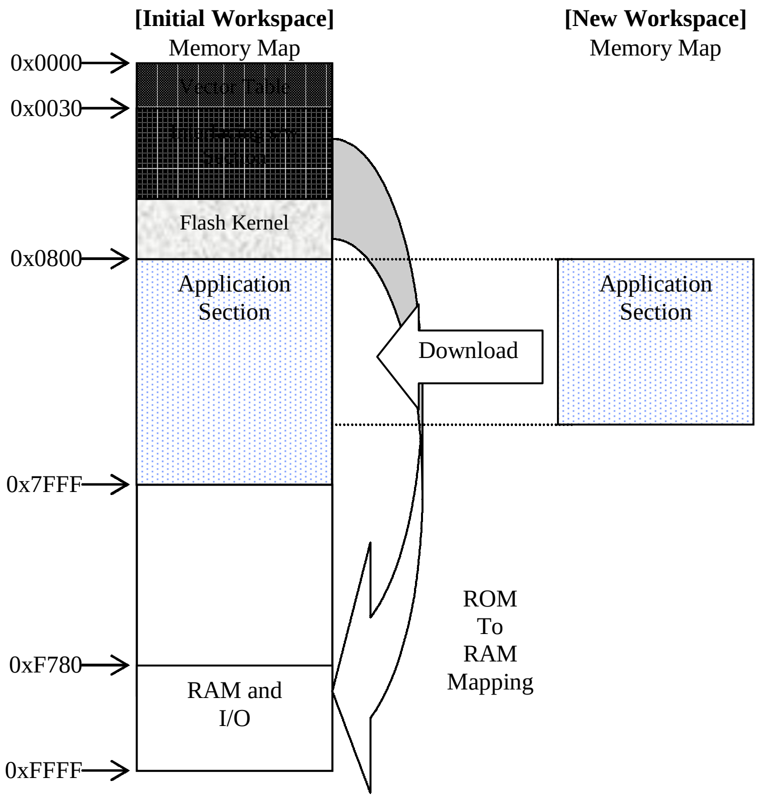


Figure 15 Memory Map for Code Upgrade Method 1

[Initial workspace]’s and [new workspace]’s working Procedure:

1. Power-up sequence
2. Enter main function
3. Initialize SCI
4. Jump to “application program”

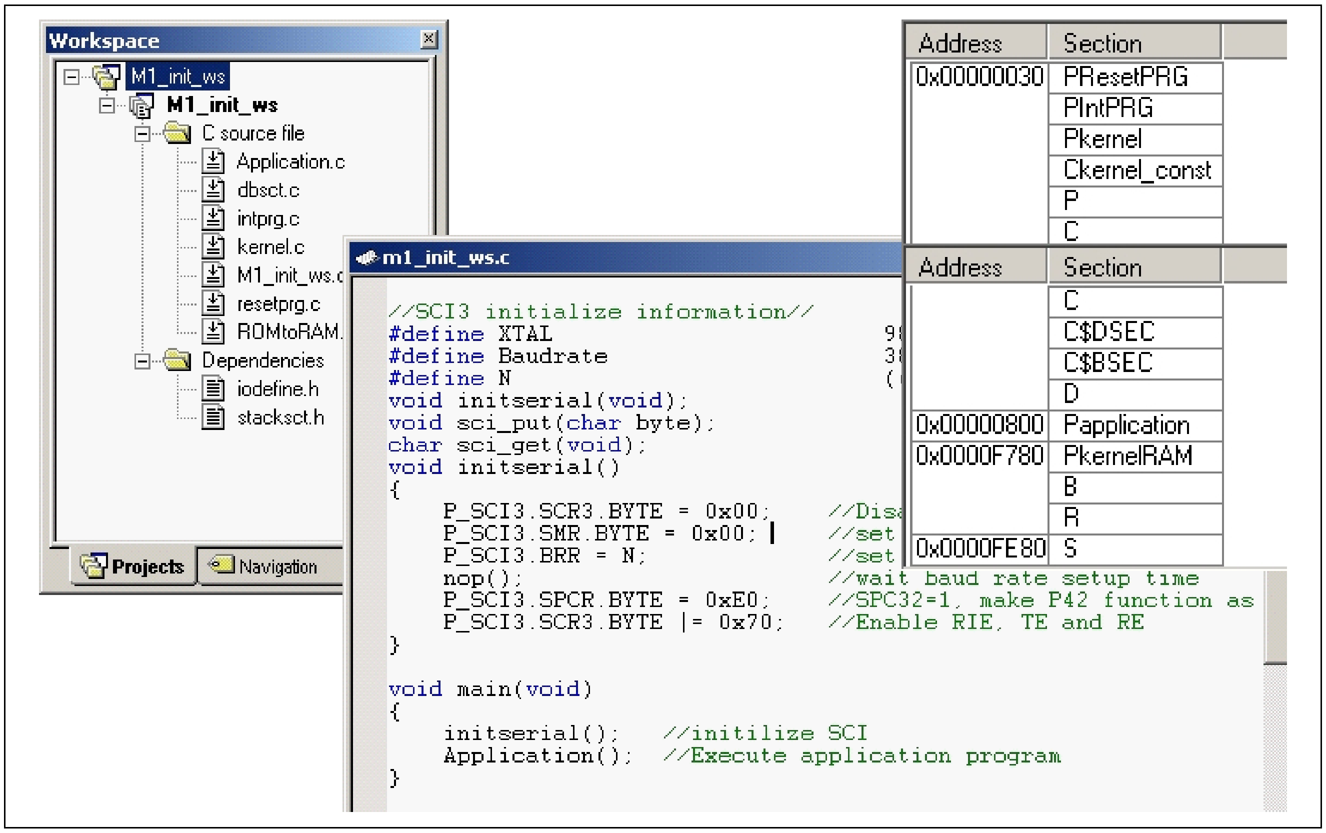
Note: This method of implementation should only be used when minor changes are made to modify the existing workspace (e.g. new function added to push button or new algorithm computation, and others value added implementation etc). There must be no changes made to the constant, variables and interrupt vector table of the initial workspace. If such changes are required, user must implement method 2 instead.

Flashing Procedure:

1. 'Download' command activated at the PC GUI.
2. SCI interrupt activated
3. MCU interface routine (UI) will jump to SCI interrupt service routine and perform:
   A. Copy flashing kernel from ROM to RAM space
   B. Obtain the data stream for flash kernel
4. Flash kernel will program the flash memory
5. Upon completion,
   A. UI will force jump to the 'Power ON Reset' function which will initialize the whole workspace, or
   B. The user may assert hardware reset signal in order to run new application, or
   C. The user may make use of watchdog timer to generate an internal reset to initialize all I/O ports to high impedance state

Steps to generate M1 [Initial workspace]:

1. Create a new workspace (application) based on SLP Toolchain
2. Write the code (and create the section name for this code)
3. Declare the section address in HEW [Option/ Toolchain/ Linker/ Section]
4. Compile to obtain the S-Record file



Address	Section
0x00000030	PResetPRG
	PIntPRG
	Pkernel
	Ckernel_const
	P
	C

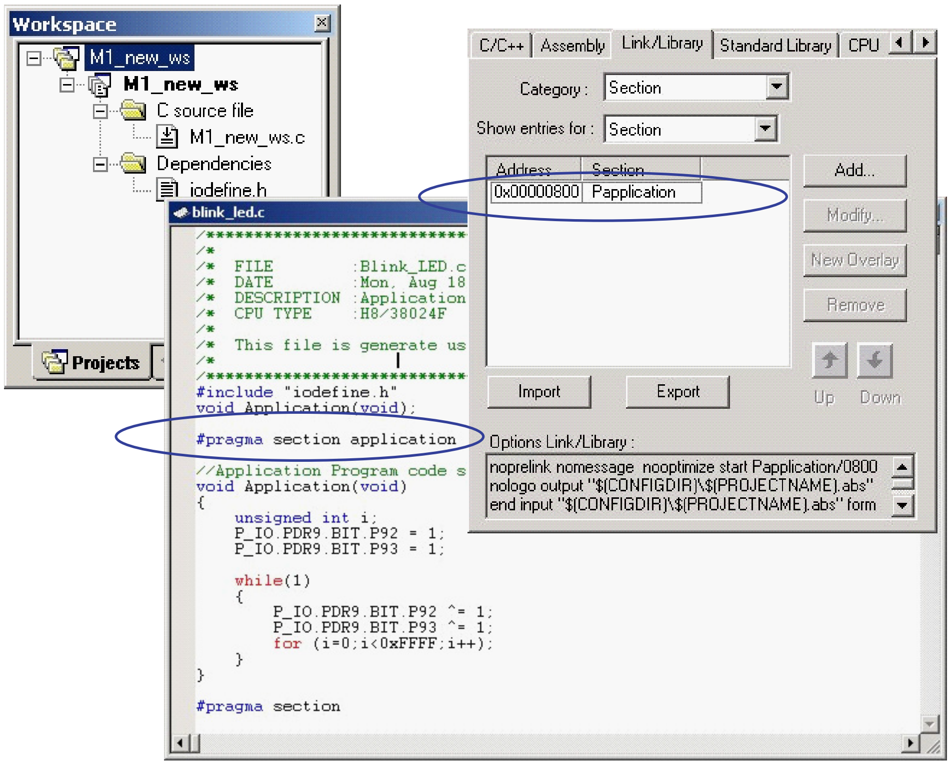
Address	Section
	C
	C$DSEC
	C$BSEC
	D
0x00000800	Papplication
0x0000F780	PkernelRAM
	B
	R
0x0000FE80	S

```
//SCI3 initialize information//
#define XTAL                                              9
#define Baudrate                                          3
#define N                                                 (
void initserial(void);
void sci_put(char byte);
char sci_get(void);
void initserial()
{
        P_SCI3.SCR3.BYTE = 0x00;     //Disa
        P_SCI3.SMR.BYTE = 0x00;      //set
        P_SCI3.BRR = N;              //set
        nop();                       //wait baud rate setup time
        P_SCI3.SPCR.BYTE = 0xE0;     //SPC32=1, make P42 function as
        P_SCI3.SCR3.BYTE |= 0x70;    //Enable RIE, TE and RE
}

void main(void)
{
        initserial();   //initilize SCI
        Application();  //Execute application program
}
```

Figure 16   Method 1 [Initial workspace] Generation

Steps to generate M1 [New Workspace]:

1. Create a new workspace (empty application) based on the SLP
2. Write the code (and create the section name for this code)
3. Declare the section address in HEW [Option/ Toolchain/ Linker/ Section]
4. Copy the iodefine.h file from the initial workspace folder to new workspace folder
   - Copy [\Method 1\M1_init_ws\M1_init_ws\iodefine.h] to [\Method 1\M1_new_ws\M1_new_ws]
5. Compile to obtain the S-Record file



```
/*****************************
/*
/*  FILE          :Blink_LED.c
/*  DATE          :Mon, Aug 18
/*  DESCRIPTION   :Application
/*  CPU TYPE      :H8/38024F
/*
/*  This file is generate us
/*
/*****************************
#include "iodefine.h"
void Application(void);

#pragma section application

//Application Program code s
void Application(void)
{
    unsigned int i;
    P_IO.PDR9.BIT.P92 = 1;
    P_IO.PDR9.BIT.P93 = 1;

    while(1)
    {
        P_IO.PDR9.BIT.P92 ^= 1;
        P_IO.PDR9.BIT.P93 ^= 1;
        for (i=0;i<0xFFFF;i++);
    }
}

#pragma section
```

Category : Section

Show entries for : Section

Address	Section
0x00000800	Papplication

Options Link/Library :

```
noprelink nomessage  nooptimize start Papplication/0800
nologo output "$(CONFIGDIR)\$(PROJECTNAME).abs"
end input "$(CONFIGDIR)\$(PROJECTNAME).abs" form
```

Figure 17 Method 1 [New workspace] generation

Highlight:

The new application has much restriction:

1. No control of interrupt entry
2. The user has to take care of copying initial data

7.2.2 Method 2 [M2]

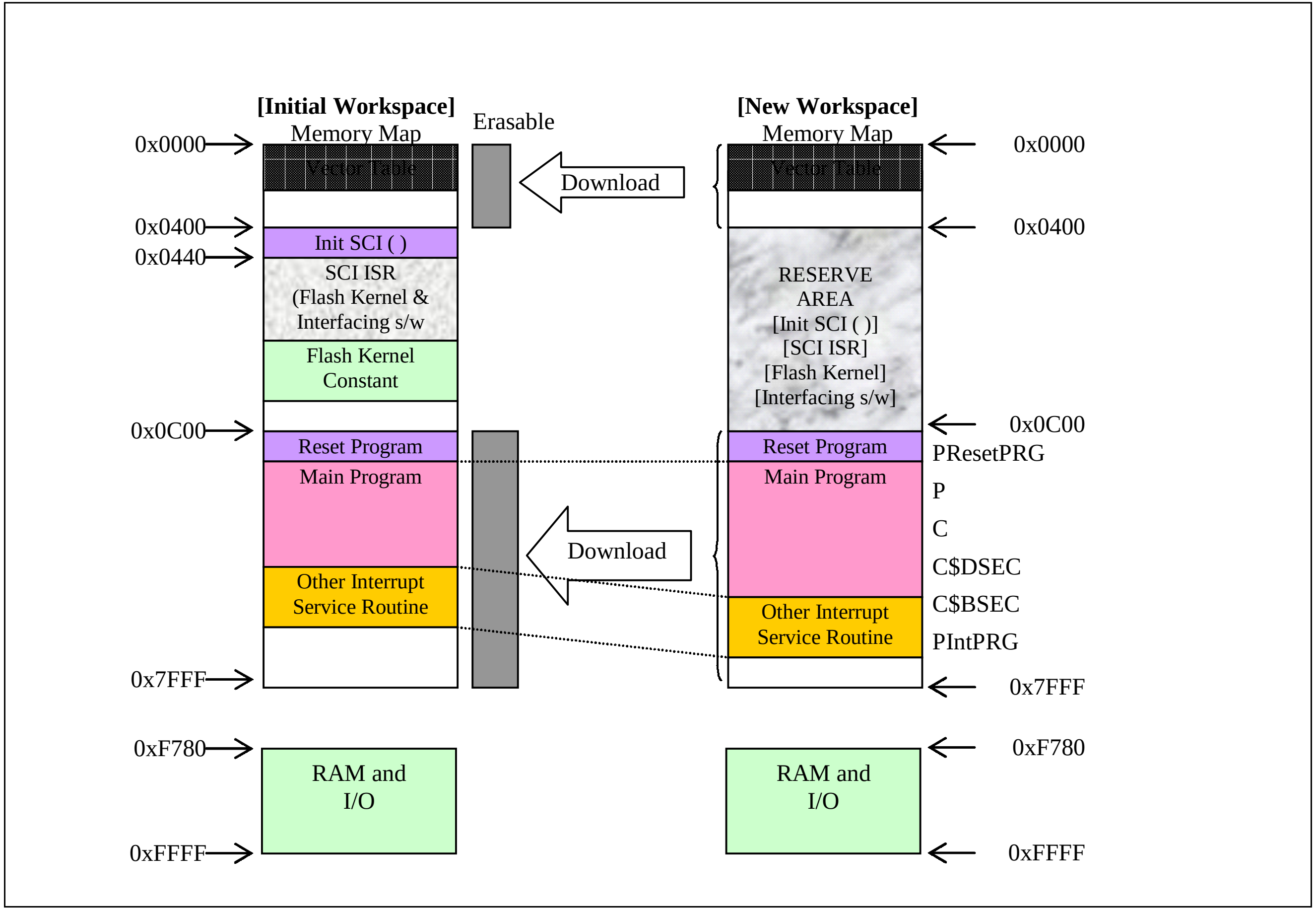


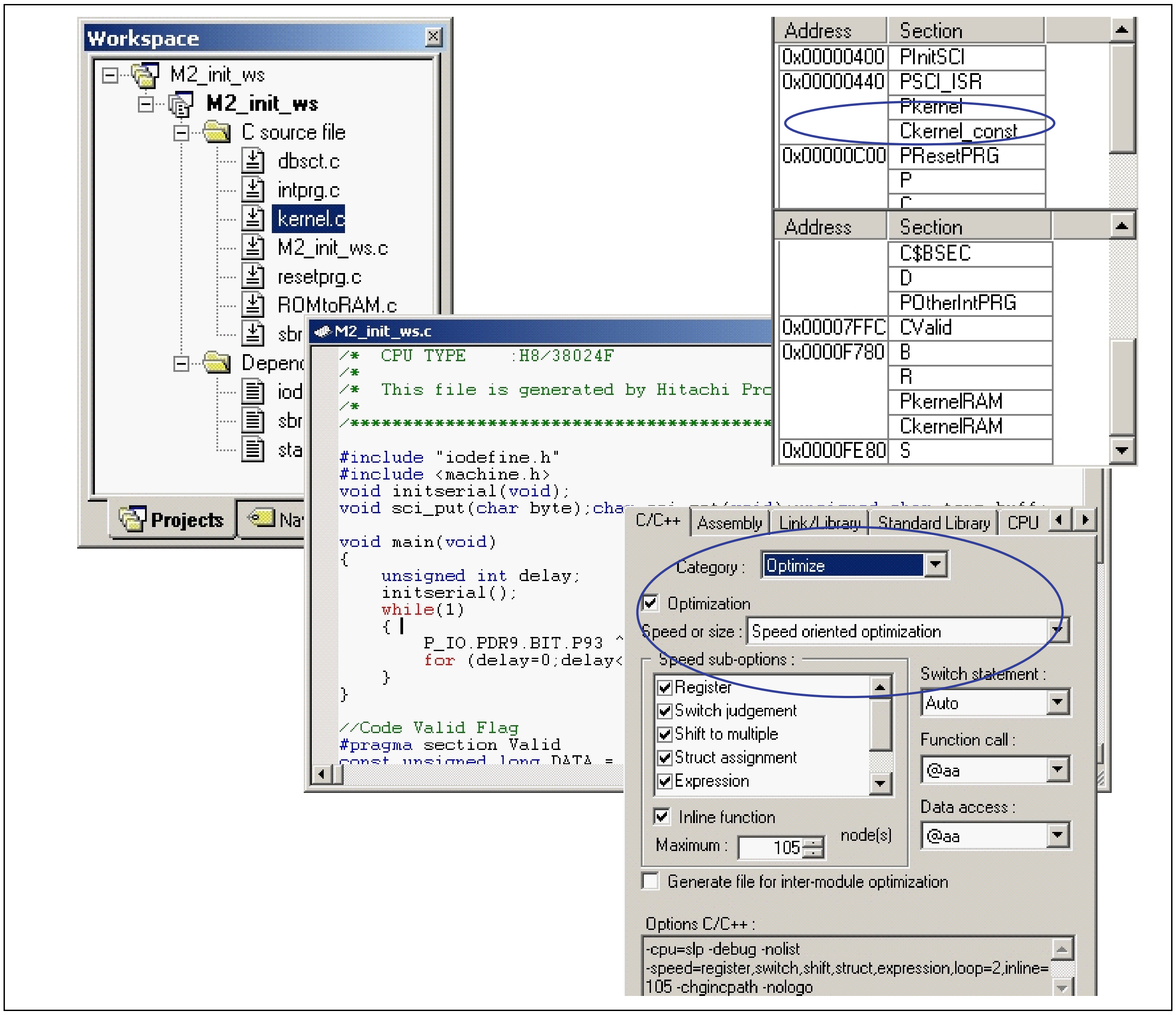
Figure 18 Memory Map for Code Upgrade Method 2

Flashing Procedure:

1. ‘Download’ command activated at the PC GUI.
2. The SCI interrupt activated
3. The MCU interface routine (UI) will jump to SCI interrupt service routine and perform:
   A. Copy flashing kernel from ROM to RAM space
   B. Obtain the data stream for flash kernel
4. The flash kernel will program the flash memory
5. Upon completion,
   A. UI will force jump to the ‘Power ON Reset’ function which will initialize the whole workspace, or
   B. The user may assert hardware reset signal in order to run new application, or
   C. The user may make use of watchdog timer to generate internal reset to initialize all I/O port to high impedance state

Steps to generate M2 [Initial workspace]:

1. Create a new workspace (application) based on SLP
2. Write the code (& create the section name for this code)
3. Declare the section address in HEW [Option/ Toolchain/ Linker/ Section]
4. Compile to obtain the S-Record file
5. Compiler setting:
   A. Optimization = Speed oriented optimization (reason is to remove "register save" library option in SCI ISR)
   B. The kernel constant section added to avoid overwriting by the [new workspace]



```
/*  CPU TYPE    :H8/38024F
/*
/*  This file is generated by Hitachi Pro
/*
/*******************************************

#include "iodefine.h"
#include <machine.h>
void initserial(void);
void sci_put(char byte);char

void main(void)
{
    unsigned int delay;
    initserial();
    while(1)
    { |
        P_IO.PDR9.BIT.P93 ^
        for (delay=0;delay<
    }
}

//Code Valid Flag
#pragma section Valid
const unsigned long DATA =
```

Address	Section
0x00000400	PInitSCI
0x00000440	PSCI_ISR
	Pkernel
	Ckernel_const
0x00000C00	PResetPRG
	P
	C

Address	Section
	C$BSEC
	D
	POtherIntPRG
0x00007FFC	CValid
0x0000F780	B
	R
	PkernelRAM
	CkernelRAM
0x0000FE80	S

Category : Optimize

☑ Optimization

Speed or size : Speed oriented optimization

Speed sub-options :
- ☑ Register
- ☑ Switch judgement
- ☑ Shift to multiple
- ☑ Struct assignment
- ☑ Expression

☑ Inline function

Maximum : 105 node(s)

Switch statement : Auto

Function call : @aa

Data access : @aa

☐ Generate file for inter-module optimization

Options C/C++ :

```
-cpu=slp -debug -nolist
-speed=register,switch,shift,struct,expression,loop=2,inline=
105 -chgincpath -nologo
```

Figure 19   Method 2 [Initial workspace] Generation

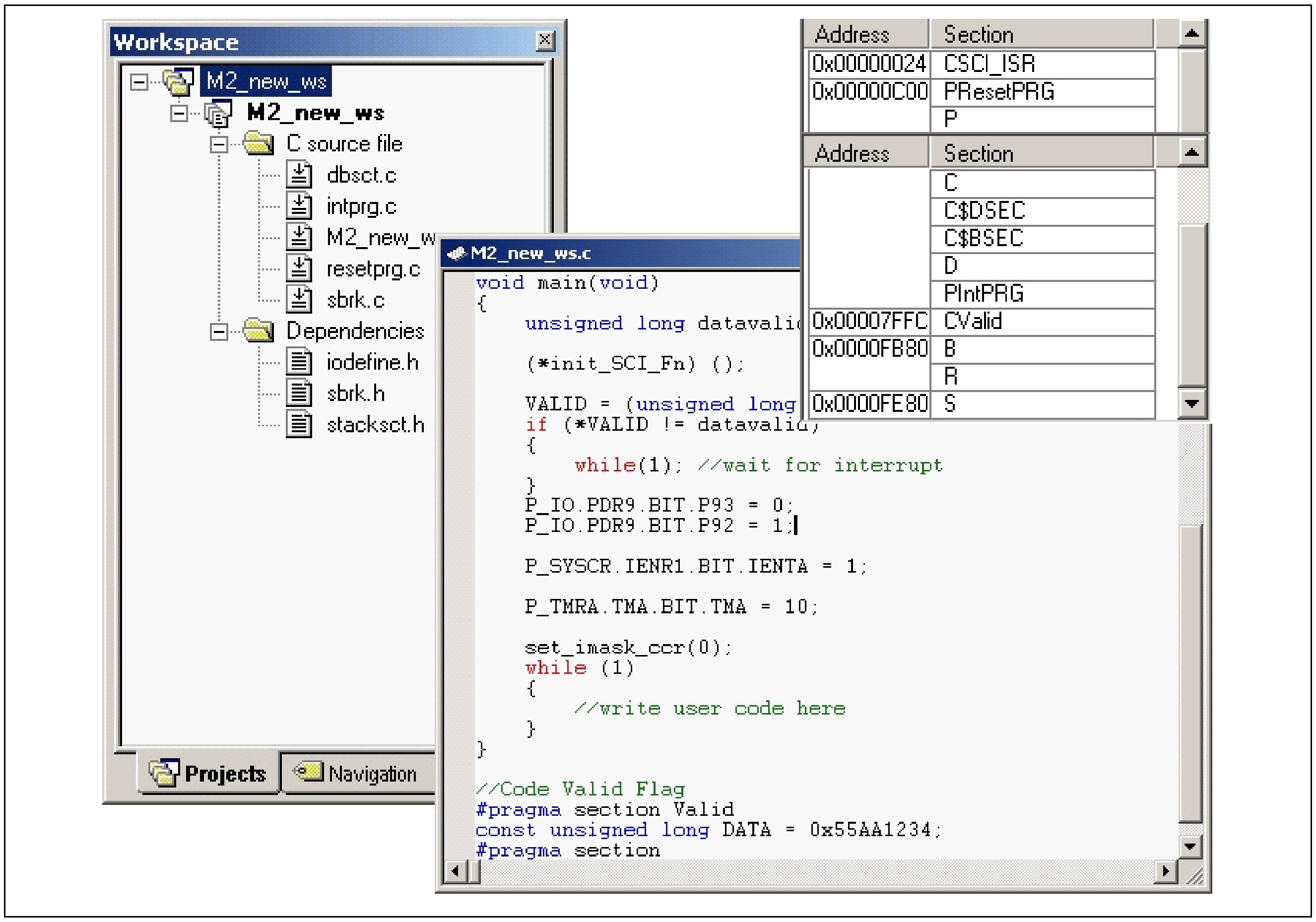
Important Note for M2 [New Workspace]:

1. Reserve H’0400 to H’0C00 (flash block 1 and 2)
   To prevent overwriting to initial application flash kernel and interfacing software
2. Fix RESET routine at H’0C00 (flash block 3)
3. Fix MAIN and other ISR after RESET routine
4. Init SCI ( ) can be access by function call to H’0400
5. SCI Interrupt Service Routine must fix at H’0440
   This can be achieve using the interrupt handler (intprg.c)
   e.g:

```
    #pragma section SCI_ISR
                   static const unsigned short DATA = 0x0440;
    #pragma section
```

Steps to Generate M2 [New Workspace]:

1. Create a new workspace (Application) based on SLP
2. Write the new workspace code
3. Declare the section address in HEW [Option/ Toolchain/ Linker/ Section]
   - please refer to the figure below for details
4. Compile to obtain the S-Record file



```
void main(void)
{
    unsigned long datavali

    (*init_SCI_Fn) ();

    VALID = (unsigned long
    if (*VALID != datavalid)
    {
        while(1); //wait for interrupt
    }
    P_IO.PDR9.BIT.P93 = 0;
    P_IO.PDR9.BIT.P92 = 1;

    P_SYSCR.IENR1.BIT.IENTA = 1;

    P_TMRA.TMA.BIT.TMA = 10;

    set_imask_ccr(0);
    while (1)
    {
        //write user code here
    }
}

//Code Valid Flag
#pragma section Valid
const unsigned long DATA = 0x55AA1234;
#pragma section
```

Address	Section
0x00000024	CSCI_ISR
0x00000C00	PResetPRG
	P

Address	Section
	C
	C$DSEC
	C$BSEC
	D
	PIntPRG
0x00007FFC	CValid
0x0000FB80	B
	R
0x0000FE80	S

Figure 20 Method 2 [New workspace] Generations

8. Overall Operation and Observations

This section shows the setup required for the application note and demonstrates the operation of the Flash GUI.

8.1 Environment Setup

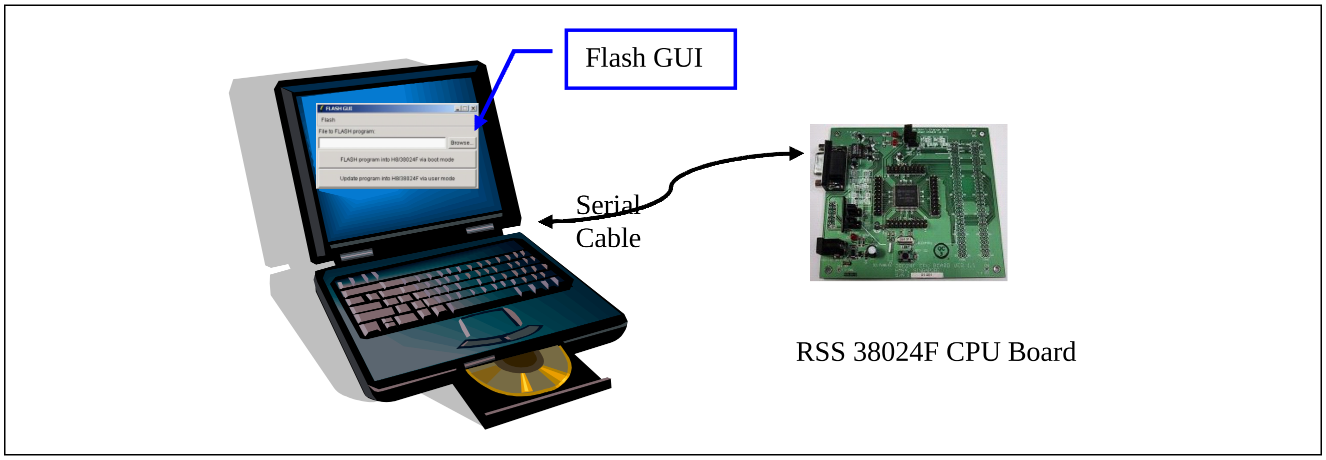


Figure 21 Environment Setup for User Mode (Re)Programming

If the RSS 38024F CPU Board is not available, a simple connection diagram is shown in the figure below:

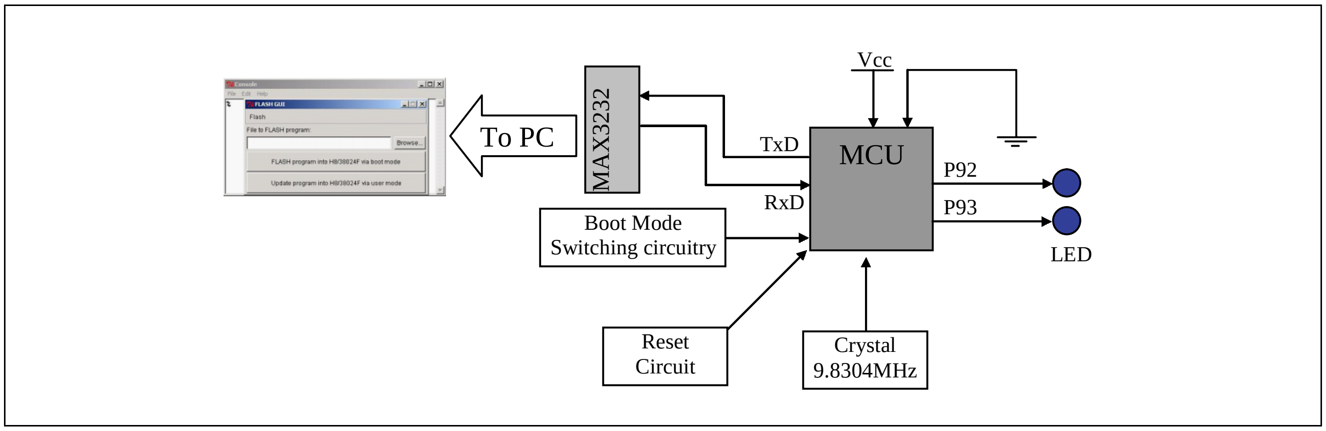


Figure 22 User Mode Programming demo board block diagram

8.2 Programming using GUI

8.2.1 Method 1 Demonstration

Boot Mode Programming:

1. Open Flash_GUI.tcl
2. Select download file “*M1_init_ws.mot*”
3. Switch the H8/38024F MCU to the boot mode* and press the reset button.
4. Click on the boot mode button, “Flash program into H8/38024F via boot mode”, on the Flash GUI to begin downloading.
5. “Program downloaded!” message box will be displayed, indicating the completion of the boot mode programming.
6. Switch the H8/38024F MCU to User Mode* and press the reset button.
7. Both of the LEDs connected to Port 9 will blink continuously indicating that the “M1_init_ws” program is running.

User Mode Programming:

1. Select *“M1_new_ws.mot*” as the input S-Record file.
2. Click “Update program into H8/38024F via user mode” to download **[New Workspace]**
3. “Program downloaded!” message box will be displayed
4. A new application program is executed causing both LEDs, D3 and D4, to light up alternately

User Mode Re-Programming:

1. Select *“M1_App1.mot”* as the input S-Record file
2. Click “Update program into H8/38024F via user mode” to download new application program

“Program downloaded!” message box will be displayed

3. A new application program is executed causing LEDs, D3 and D4, to blink together

The user is able to download and execute different application programs in User mode without resetting MCU.

Note: *Refer to 38024F CPU Board Quick Start Guide for jumper settings to switch to Boot Mode and User Mode

8.2.2 Method 2 Demonstration

The Method 2 demonstration can be access by repeat section 8.2.1 and change the downloading file name:

e.g.:

“M1_init_ws.mot”	➔	*“M2_init_ws.mot*”
“M1_new_ws.mot”	➔	*“M2_new_ws.mot*”
“M1_APP1.mot”	➔	*“M2_APP1.mot*”

The result of the demonstration is same.

9. Code Listing

The attached code is generated using HEW project generator targeting at the H8/38024F SLP MCU. The toolchain used is the free SLP/Tiny toolchain.

9.1 Method 1 [Initial Workspace] Code Listing

9.1.1 M1 [Initial Workspace] Main Routine

The Figure below shows the flow chart for “m1_init_ws.c”, followed by its code listing.

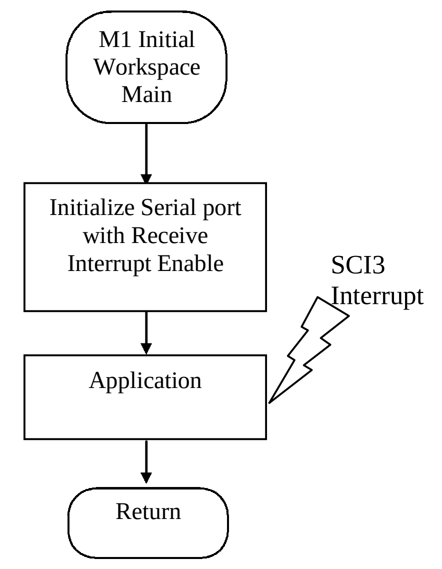

M1 Initial Workspace Main
Initialize Serial port with Receive Interrupt Enable
SCI3 Interrupt
Application
Return

Figure 23 Flow Chart for M1 [Initial Workspace] Main Routine

```
/**********************************************************************/
/*                                                                    */
/*  FILE        :M1_init_ws.c                                         */
/*  DATE        :Mon, Sep 29, 2003                                    */
/*  DESCRIPTION :Main Program                                         */
/*  CPU TYPE    :H8/38024F                                            */
/*                                                                    */
/*  This file is generated by Renesas Project Generator (Ver.2.1).    */
/*                                                                    */
/**********************************************************************/
#include "iodefine.h"
#include <machine.h>

//Flash function prototype
extern void copyfunc(void);
extern unsigned char prog_flash_line_128 (unsigned long t_address, union
char_rd_datum_union *p_data);
extern unsigned char erase_block (unsigned char block_num);
extern int  *_PkernelBegin, *_PkernelEnd, *_Pkernel_RAMBegin;
extern int  *_CkernelBegin, *_CkernelEnd, *_Ckernel_RAMBegin;

//function prototype
void copyfunc(void);
extern void Application(void);

//SCI3 initialize information//
#define XTAL                          9830400L
#define Baudrate                      38400L
#define N                             ((XTAL) / (64L*1L*Baudrate)) - 1L
void initserial(void);
void sci_put(char byte);
char sci_get(void);
void initserial()
{
   P_SCI3.SCR3.BYTE = 0x00;  //Disable TIE,TE,RE,MPIE,TEIE,RIE,
   P_SCI3.SMR.BYTE = 0x00;   //set Async, 8 data, none parity, 1 stop, clk n=0
   P_SCI3.BRR = N;           //set baud rate = N
   nop();                    //wait baud rate setup time
   P_SCI3.SPCR.BYTE = 0xE0;  //SPC32=1, make P42 function as TXD32
   P_SCI3.SCR3.BYTE |= 0x70; //Enable RIE, TE and RE
}

void main(void)
{
   initserial();             //initilize SCI
   Application();            //Execute application program
}
```

```
void copyfunc(void)
{
   register int *p, *q;
   for (p=_PkernelBegin, q=_Pkernel_RAMBegin;p<_PkernelEnd;p++,q++)
   {
      *q=*p;
   }

   for (p=_CkernelBegin, q=_Ckernel_RAMBegin;p<_CkernelEnd;p++,q++)
   {
      *q=*p;
   }
}

void sci_put(char byte)
{
   while(P_SCI3.SSR.BIT.TDRE==0);
   P_SCI3.TDR=byte;
   while(P_SCI3.SSR.BIT.TEND==0);
}

char sci_get(void)
{
   while(P_SCI3.SSR.BIT.RDRF==0){}       //Wait until RDRF = 1
   if ((P_SCI3.SSR.BYTE & 0x38) ==0)     //Check for SCI error
   {
   return P_SCI3.RDR;
   }
   else return 0xFF;                     //If error occur return 0xFF
   if(P_SCI3.SSR.BIT.RDRF==1) P_SCI3.SSR.BIT.RDRF=0;
}

#pragma section
```

9.1.2 M1 [Initial Workspace] Application Routine

The figure below shows the flow chart for “Application.c”, followed by its code listing.

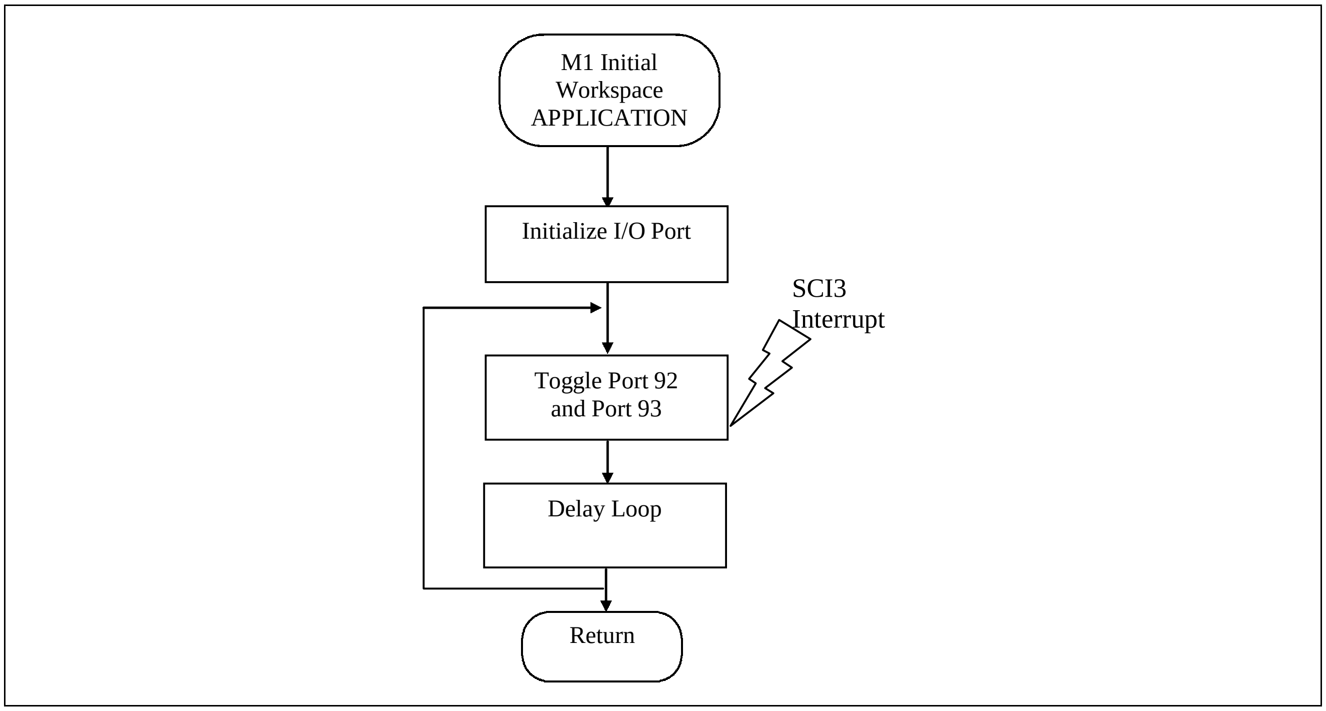


Figure 24 Flow Chart for M1 [Initial Workspace] Application Routine

```
#include "iodefine.h"
//Section define for application program
#pragma section application
void Application(void);
//Application Program code start
//Blinking LED application
void Application(void)
{
   unsigned int i;
   P_IO.PDR9.BIT.P92 = 1;
   P_IO.PDR9.BIT.P93 = 1;

   while(1)
   {
      P_IO.PDR9.BIT.P92 ^= 1;
      P_IO.PDR9.BIT.P93 ^= 1;
      for (i=0;i<0xFFFF;i++);
   }
}
#pragma section
```

9.1.3 M1 [Initial Workspace] Interrupt Routine

The figure below shows the flow chart for the SCI interrupt service routine, followed by its code listing.

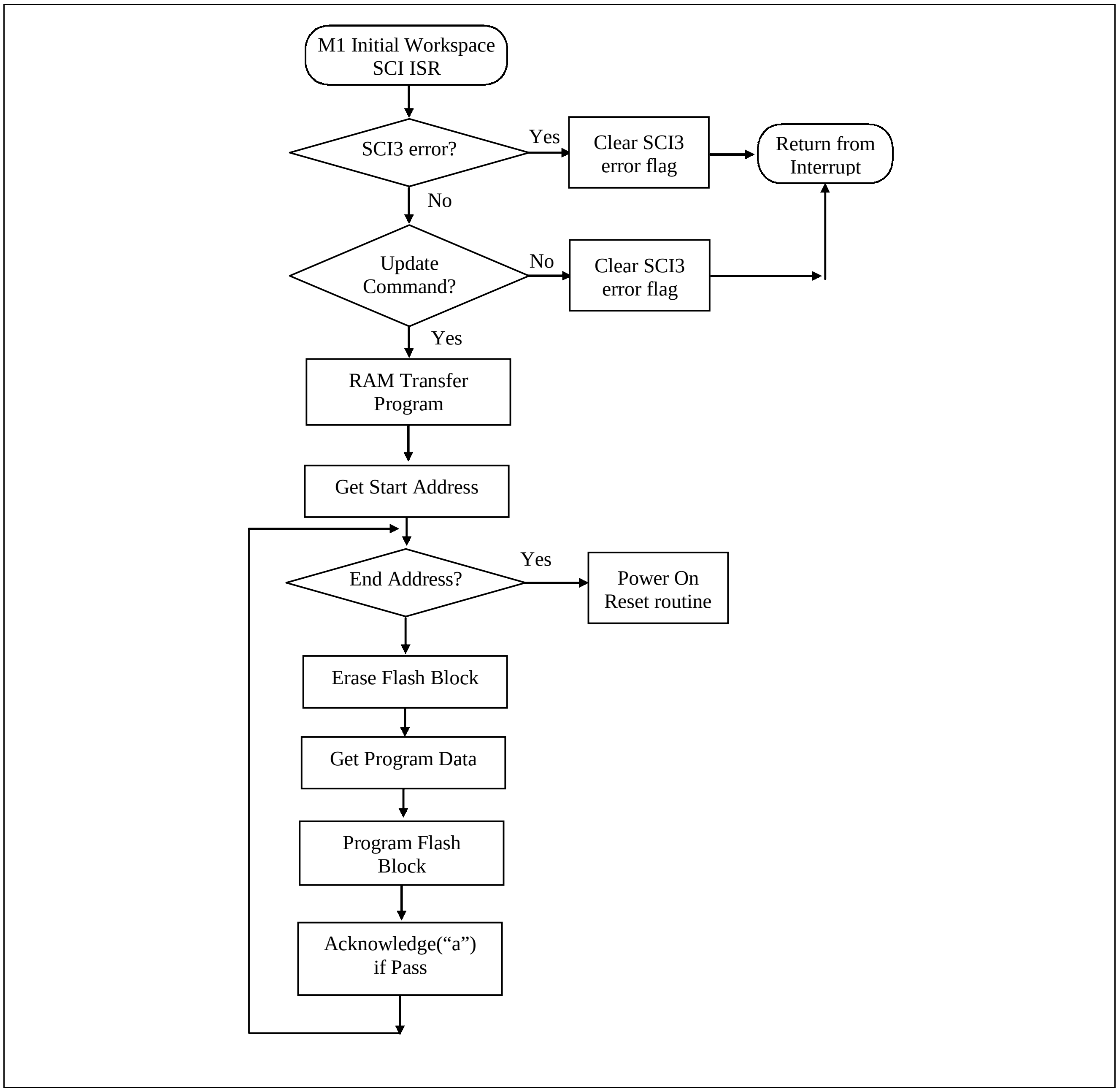


Figure 25 Flow Chart for M1 [Initial Workspace] SCI Interrupt Service Routine

```
/*********************************************************************/
/*                                                                   */
/*  FILE        :intprg.c                                            */
/*  DATE        :Mon, Sep 29, 2003                                   */
/*  DESCRIPTION :Interrupt Program                                   */
/*  CPU TYPE    :H8/38024F                                           */
/*                                                                   */
/*  This file is generated by Renesas Project Generator (Ver.2.1).   */
/*                                                                   */
/*********************************************************************/
#include "iodefine.h"
#include <machine.h>

//SCI function prototype
extern void sci_put(char byte);
extern char sci_get(void);
extern unsigned char temp_buff;

//Flash function prototype
extern unsigned char prog_flash_line_128 (unsigned long t_address, union
char_rd_datum_union *p_data);
extern unsigned char erase_block (unsigned char block_num);
extern void PowerON_Reset(void);

#pragma section IntPRG
// vector 1 Reserved
.
.
__interrupt(vect=16) void INT_TimerG(void) {/* sleep(); */}
// vector 17 Reserved

// vector 18 SCI3
__interrupt(vect=18) void INT_SCI3(void)
{
   unsigned short start_address;
   unsigned char prog_data_addr[128],count1;
   unsigned char temp_buff;
   if ((P_SCI3.SSR.BYTE & 0x38) == 0)    //Check for SCI error
   {
      if(P_SCI3.RDR=='U')
      {
            copyfunc();
            while(1)
            {
               //GET START ADDRESS
               temp_buff = sci_get();
               start_address = (unsigned short) (temp_buff <<8); //high byte
               sci_put(temp_buff);

               temp_buff = sci_get();
               start_address = start_address | (unsigned short) (temp_buff);
                                                                  //low byte
               sci_put(temp_buff);
               if    (start_address == 0x0000)      {erase_block (0);}
               else if (start_address == 0x0400)    {erase_block (1);}
               else if (start_address == 0x0800)    {erase_block (2);}
               else if (start_address == 0x0c00)    {erase_block (3);}
               else if (start_address == 0x1000)    {erase_block (4);}
               else if (start_address == 0x8000)    {PowerON_Reset();}
                                                    //end of flash programming
               else  nop(); //invalid start address

               for(count1=0;count1<128;count1++)
               {
                  prog_data_addr[count1] =  sci_get();
               }

               if(prog_flash_line_128 (start_address, (union
                     char_rd_datum_union * ) prog_data_addr)==0x01)
               {
                  sci_put('a');
               }
               else sci_put('n');
            }

         }
         else  return; // if not Update flash command then do nothing
      }
      else
      {
         //SCI error occur
         if (P_SCI3.SSR.BIT.OER == 1)
         temp_buff = P_SCI3.RDR;
         temp_buff = P_SCI3.RDR;
         P_SCI3.SSR.BYTE=0x84;
         sci_put('e');
      }
   }
   // vector 19 ADI
   __interrupt(vect=19) void INT_ADI(void) {/* sleep(); */}
   // vector 20 Direct Transition
   __interrupt(vect=20) void INT_Direct_Transition(void) {/* sleep(); */}
```

9.2 Method 1 [New Workspace] Code Listing

M1 [New Workspace] Application Routine:

The figure below shows the flow chart for “m1_new_ws.c”, followed by its code listing.

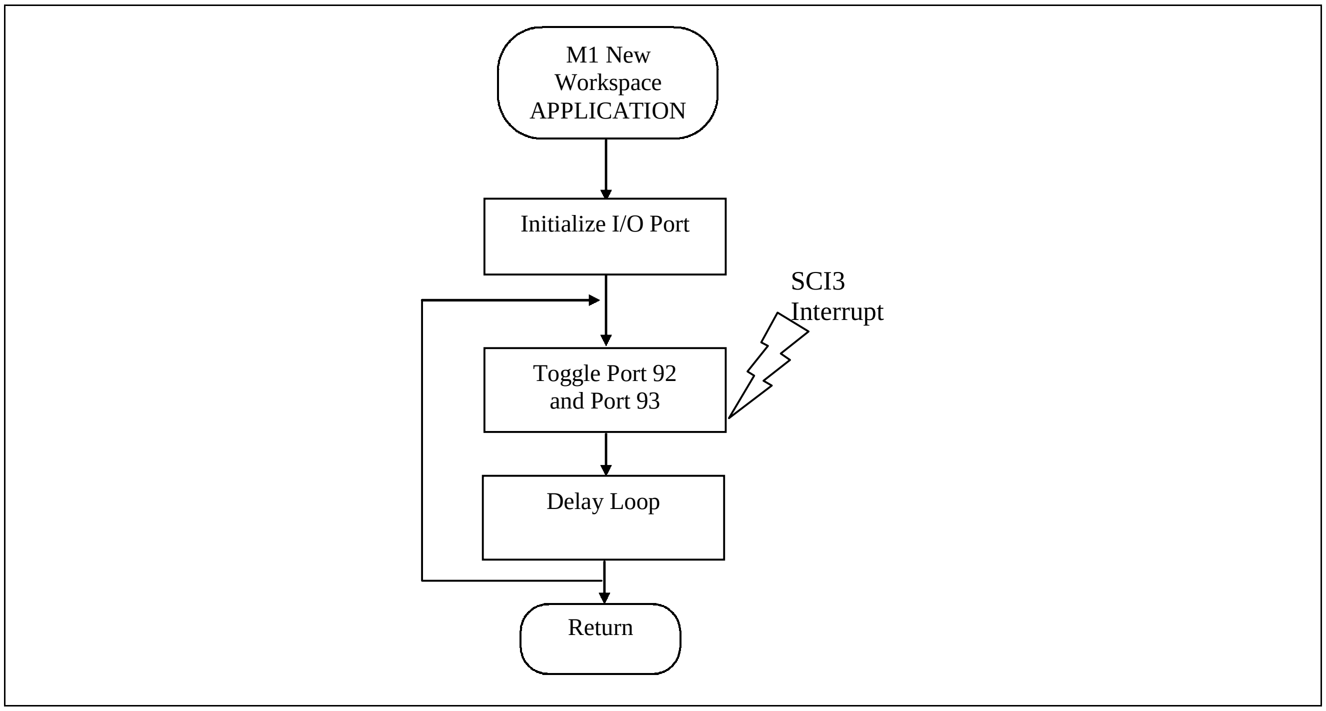


Figure 26 Flow Chart for M1 [New Workspace] Application Routine

```
#include "iodefine.h"
void Application(void);

#pragma section application

//Application Program code start
void Application(void)
{
   unsigned int i;
   P_IO.PDR9.BIT.P92 = 1;
   P_IO.PDR9.BIT.P93 = 0;

   while(1)
   {
      P_IO.PDR9.BIT.P92 ^= 1;
      P_IO.PDR9.BIT.P93 ^= 1;
      for (i=0;i<0xFFFF;i++);
   }
}

#pragma section
```

9.3 Method 2 [Initial Workspace] Code Listing

9.3.1 M2 [Initial Workspace] Main Routine

The figure below shows the flow chart for “m2_init_ws.c”, followed by its code listing.

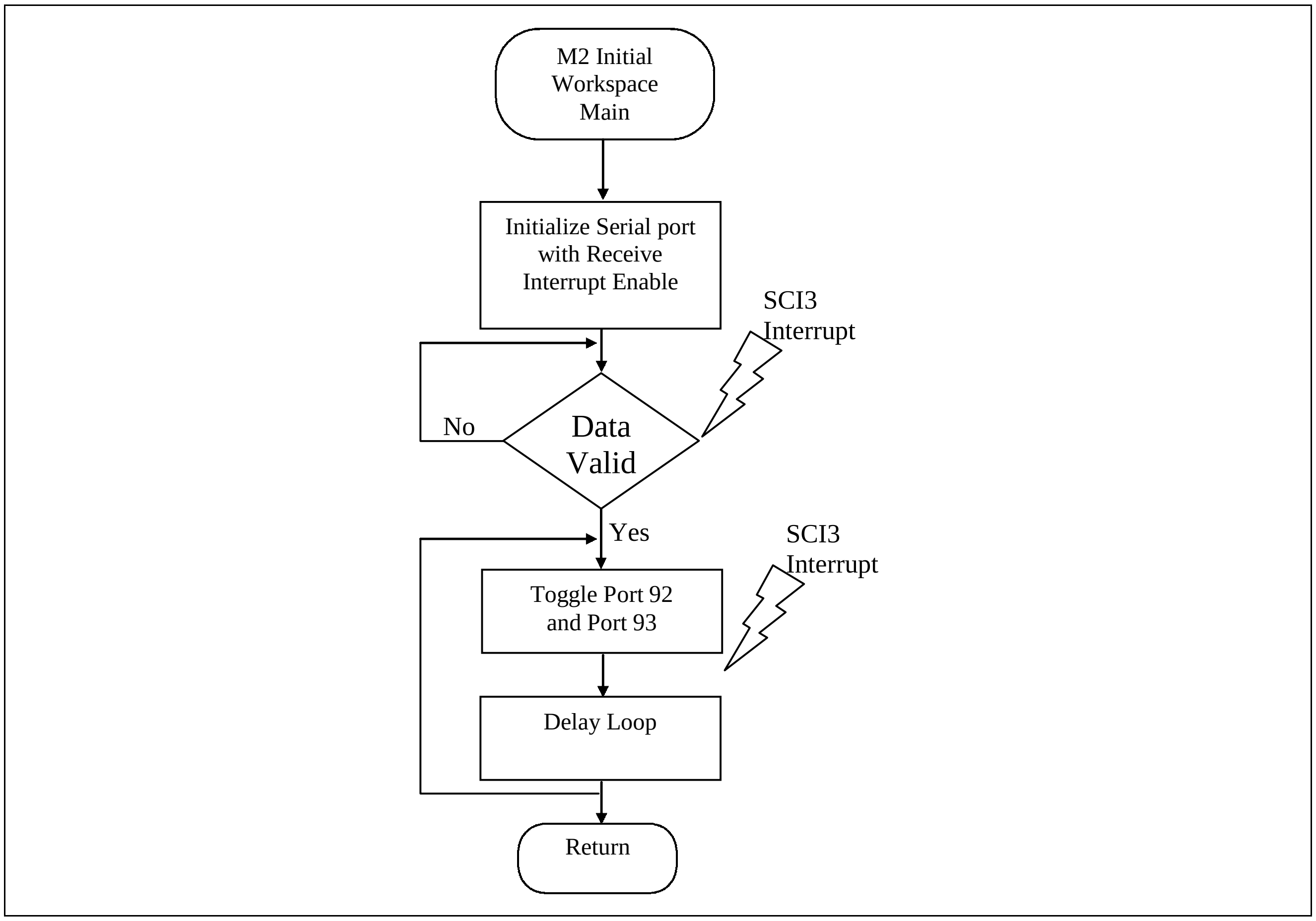

M2 Initial Workspace Main
Initialize Serial port with Receive Interrupt Enable
SCI3 Interrupt
No
Data Valid
Yes
SCI3 Interrupt
Toggle Port 92 and Port 93
Delay Loop
Return

Figure 27 Flow Chart for M2 [Initial Workspace] Main Routine

```
/***********************************************************************/
/*                                                                     */
/*  FILE        :M2_init_ws.c                                          */
/*  DATE        :Mon, Sep 29, 2003                                     */
/*  DESCRIPTION :Main Program                                          */
/*  CPU TYPE    :H8/38024F                                             */
/*                                                                     */
/*  This file is generated by Renesas Project Generator (Ver.2.1).     */
/*                                                                     */
/***********************************************************************/

#include "iodefine.h"
#include <machine.h>
void initserial(void);
void sci_put(char byte);char sci_get(void);unsigned char temp_buff;

void main(void)
{
   unsigned int delay;
   unsigned long datavalid = 0x55AA1234, *VALID;

   initserial();
   VALID = (unsigned long *)0x7FFC;
   if (*VALID != datavalid)
   {
      while(1); //wait for interrupt
   }
   P_IO.PDR9.BIT.P93 = 1;
   P_IO.PDR9.BIT.P92 = 1;

   while(1)
   {
      P_IO.PDR9.BIT.P93 ^= 1;
      P_IO.PDR9.BIT.P92 ^= 1;
      for (delay=0;delay<0xFFFF;delay++);
   }
}

//Code Valid Flag fixed at last address (0x7FFC-0x7FFF)
#pragma section Valid
const unsigned long DATA = 0x55AA1234;
#pragma section
```

```
//Init SCI routine fixed at address 0x0400
#pragma section InitSCI
//SCI3 initialize information
#define XTAL                    9830400L
#define Baudrate                38400L
#define N                       ((XTAL) / (64L*1L*Baudrate)) - 1L

//unsigned char *addr, temp;
void initserial()
{
   P_SCI3.SCR3.BYTE = 0x00;  //Disable TIE,TE,RE,MPIE,TEIE,RIE,
   P_SCI3.SMR.BYTE = 0x00;         //set Async, 8 data, none parity, 1 stop,
clk n=0
   P_SCI3.BRR = N;                 //set baud rate = 9600
   nop();                    //wait baud rate setup time
   P_SCI3.SPCR.BYTE = 0xE0;  //SPC32=1, make P42 function as TXD32
   P_SCI3.SCR3.BYTE |= 0x70; //Enable RIE, TE and RE
   set_imask_ccr(0);
}
//SCI3 initialize information end//
#pragma section
```

9.3.2 M2 [Initial Workspace] Interrupt Routine

The figure below shows the flow chart for “m2_init_ws.c”, followed by its code listing.

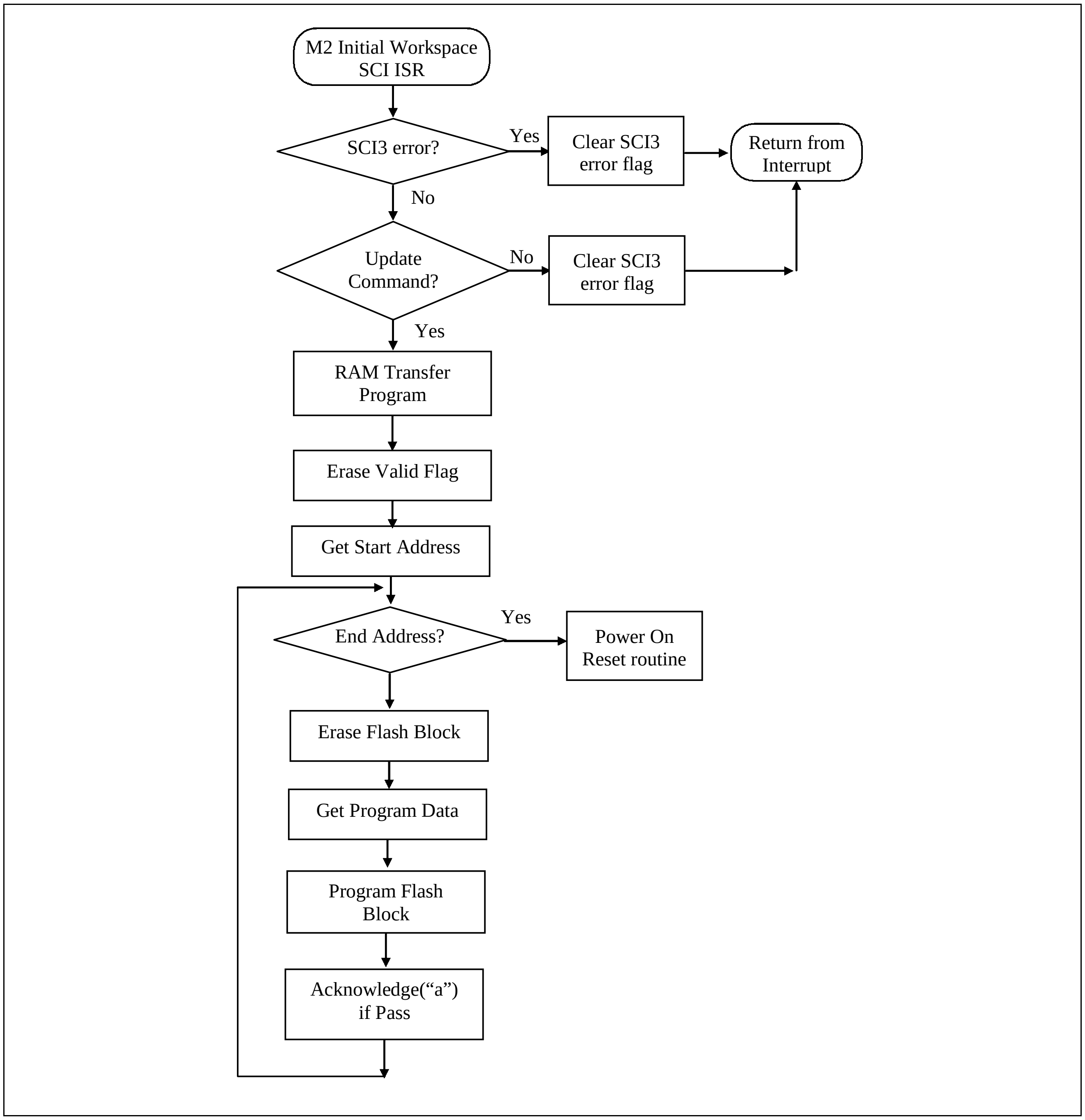


Figure 28 Flow Chart for M2 [Initial Workspace] SCI Interrupt Service Routine

```
/***********************************************************************/
/*                                                                     */
/*  FILE        :intprg.c                                              */
/*  DATE        :Mon, Sep 29, 2003                                     */
/*  DESCRIPTION :Interrupt Program                                     */
/*  CPU TYPE    :H8/38024F                                             */
/*                                                                     */
/*  This file is generated by Renesas Project Generator (Ver.2.1).     */
/*                                                                     */
/***********************************************************************/
#include "iodefine.h"
#include <machine.h>

//SCI function prototype
void sci_put(char byte);char sci_get(void);
extern unsigned char temp_buff;

//Flash function prototype
void copyfunc(void);
extern unsigned char prog_flash_line_128 (unsigned long t_address, union
char_rd_datum_union *p_data);
extern unsigned char erase_block (unsigned char block_num);
extern int  *_PkernelBegin, *_PkernelEnd, *_Pkernel_RAMBegin;
extern int  *_CkernelBegin, *_CkernelEnd, *_Ckernel_RAMBegin;

extern void PowerON_Reset(void);

#pragma section OtherIntPRG
// vector 1 Reserved
.
.

// vector 19 ADI
__interrupt(vect=19) void INT_ADI(void) {/* sleep(); */}
// vector 20 Direct Transition
__interrupt(vect=20) void INT_Direct_Transition(void) {/* sleep(); */}

//SCI ISR section fixed at 0x0440
#pragma section SCI_ISR
// vector 18 SCI3
__interrupt(vect=18) void INT_SCI3(void)
{

   unsigned short start_address;
   unsigned char prog_data_addr[128],count1;

   if ((P_SCI3.SSR.BYTE & 0x38) == 0)    //Check for SCI error
   {
      if(P_SCI3.RDR=='U')
      {
            copyfunc();

            erase_block (4); //erase Valid Flag
            while(1)
            {
               //GET START ADDRESS
               temp_buff = sci_get();
               start_address = (unsigned short) (temp_buff <<8); //high byte
               sci_put(temp_buff);

               temp_buff = sci_get();
               start_address = start_address | (unsigned short) (temp_buff);
                                                                 //low byte
               sci_put(temp_buff);

               if     (start_address == 0x0000) {erase_block (0);}
               else if (start_address == 0x0400) {erase_block (1);}
               else if (start_address == 0x0800) {erase_block (2);}
               else if (start_address == 0x0c00) {erase_block (3);}
               else if (start_address == 0x1000) {erase_block (4);}
               else if (start_address == 0x8000)
                     {PowerON_Reset();}//end of flash programming
               else  nop(); //invalid start address

               for(count1=0;count1<128;count1++)
               {
                  prog_data_addr[count1] =  sci_get();
               }
               if(prog_flash_line_128 (start_address, (union
                     char_rd_datum_union * ) prog_data_addr)==0x01)
               {
                  sci_put('a');
               }
               else sci_put('n');
            }
         }
         else  return; // if not Update flash command then do nothing
      }
      else
      {
         //SCI error occur
         if (P_SCI3.SSR.BIT.OER == 1)
         temp_buff = P_SCI3.RDR;
         temp_buff = P_SCI3.RDR;
         P_SCI3.SSR.BYTE=0x84;
         sci_put('e');
      }
   }
```

```
void sci_put(char byte)
{
   while(P_SCI3.SSR.BIT.TDRE==0){}
   P_SCI3.TDR=byte;
   while(P_SCI3.SSR.BIT.TEND==0){}
}

char sci_get(void)
{
   while(P_SCI3.SSR.BIT.RDRF==0){}       //Wait until RDRF = 1
   if ((P_SCI3.SSR.BYTE & 0x38) ==0)     //Check for SCI error
   {
   return P_SCI3.RDR;
   }
   else return 0xFF;                     //If error occur return 0xFF
   if(P_SCI3.SSR.BIT.RDRF==1) P_SCI3.SSR.BIT.RDRF=0;
}

void copyfunc(void)
{
   register int *p, *q;
   for (p=_PkernelBegin, q=_Pkernel_RAMBegin;p<_PkernelEnd;p++,q++){*q=*p;}
   for (p=_CkernelBegin, q=_Ckernel_RAMBegin;p<_CkernelEnd;p++,q++){*q=*p;}
}
#pragma section
```

9.4 Method 2 [New Workspace] Code Listing

9.4.1 M2 [New Workspace] Main Routine

The figure below shows the flow chart for “m2_new_ws.c”, followed by its code listing.

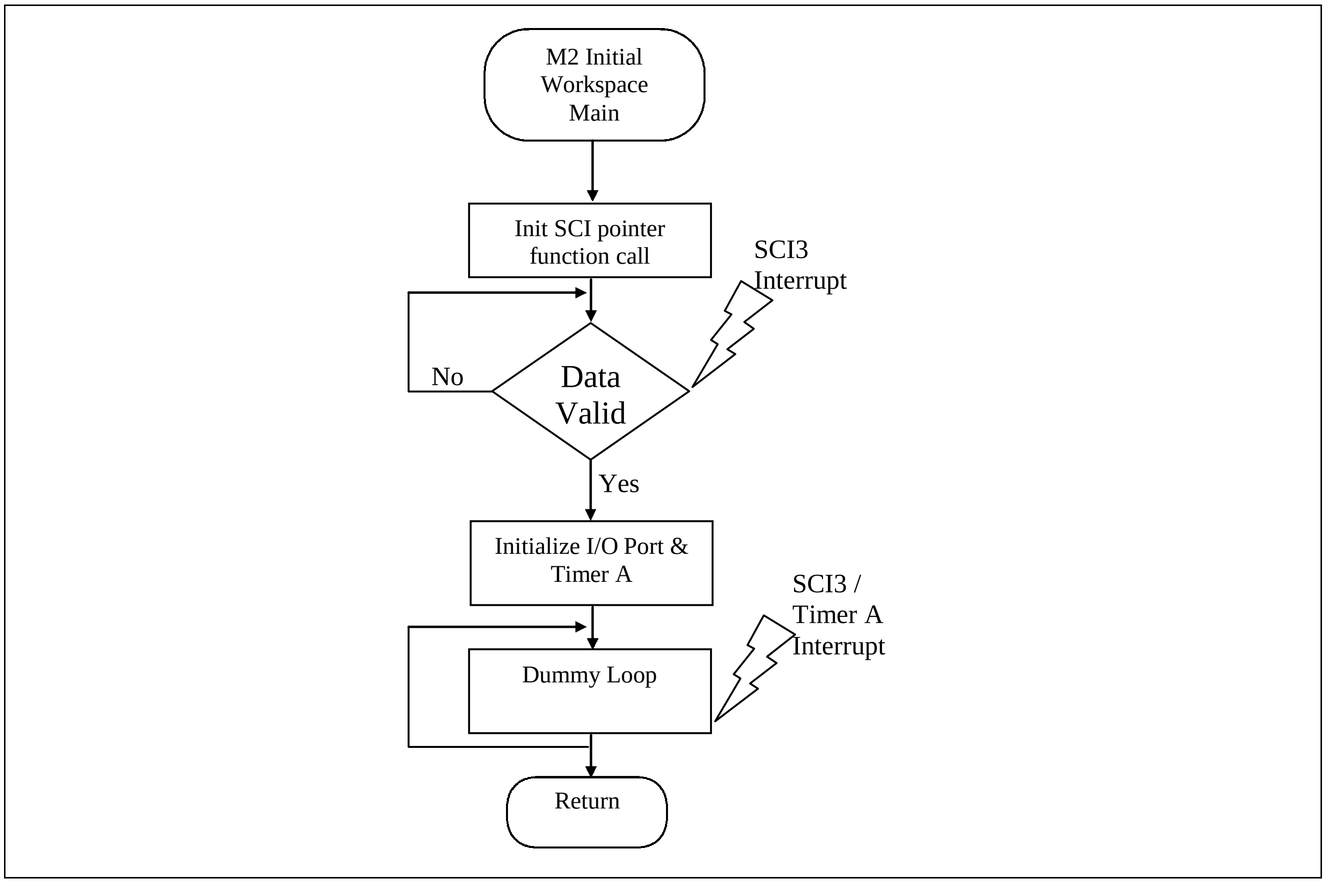

M2 Initial Workspace Main
Init SCI pointer function call
SCI3 Interrupt
Data Valid
No
Yes
Initialize I/O Port & Timer A
SCI3 / Timer A Interrupt
Dummy Loop
Return

Figure 29 Flow Chart for M2 [New Workspace] Main Routine

```
/***********************************************************************/
/*                                                                     */
/*  FILE        :M2_new_ws.c                                           */
/*  DATE        :Mon, Sep 29, 2003                                     */
/*  DESCRIPTION :Main Program                                          */
/*  CPU TYPE    :H8/38024F                                             */
/*                                                                     */
/*  This file is generated by Renesas Project Generator (Ver.2.1).     */
/*                                                                     */
/***********************************************************************/
#include "iodefine.h"
#include <machine.h>

//pointer function call to init SCI
typedef  void          (*init_SCI_FnPtr)(void);
#define init_SCI_Fn    (init_SCI_FnPtr)((unsigned short *)(0x0400))

void main(void)
{
   unsigned long datavalid = 0x55AA1234, *VALID;
   unsigned int delay = 0;

   (*init_SCI_Fn) ();

   VALID = (unsigned long *)0x7FFC;
   if (*VALID != datavalid)
   {
      while(1); //wait for interrupt
   }
   P_IO.PDR9.BIT.P93 = 1;
   P_IO.PDR9.BIT.P92 = 1;

   P_SYSCR.IENR1.BIT.IENTA = 1;

   P_TMRA.TMA.BIT.TMA = 10;

   set_imask_ccr(0);
   while (1)
    {
      //write user code here
   }
}
//Code Valid Flag
#pragma section Valid
const unsigned long DATA = 0x55AA1234;
#pragma section
```

9.4.2 M2 [New Workspace] Interrupt Routine

The figure below shows the flow chart for “intprg.c”, followed by its code listing.

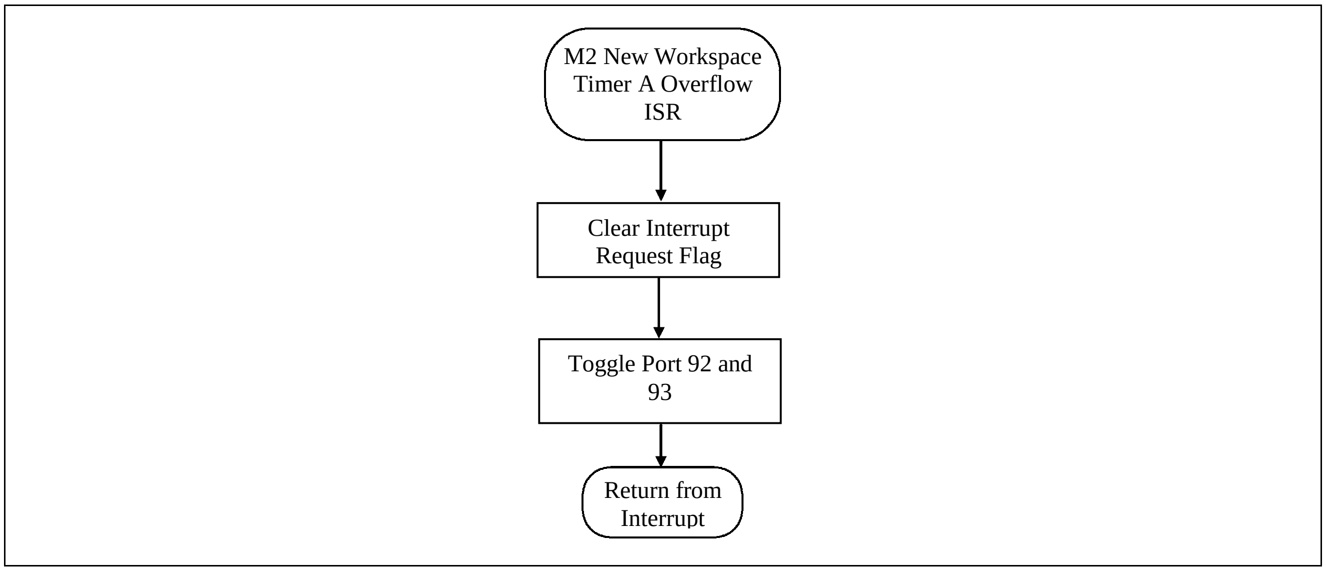

M2 New Workspace Timer A Overflow ISR
Clear Interrupt Request Flag
Toggle Port 92 and 93
Return from Interrupt

Figure 30 The Flow Chart for M2 [New Workspace] Timer A Interrupt Service Routine

```
/***********************************************************************/
/*                                                                     */
/*  FILE        :intprg.c                                              */
/*  DATE        :Mon, Sep 29, 2003                                     */
/*  DESCRIPTION :Interrupt Program                                     */
/*  CPU TYPE    :H8/38024F                                             */
/*                                                                     */
/*  This file is generated by Renesas Project Generator (Ver.2.1).    */
/*                                                                     */
/***********************************************************************/
#include "iodefine.h"
#include <machine.h>
#pragma section IntPRG
// vector 1 Reserved
.
.
// vector 10 Reserved

// vector 11 Timer A Overflow
__interrupt(vect=11) void INT_TimerA(void)
{
   unsigned int delay = 0;
   if (P_SYSCR.IRR1.BIT.IRRTA == 1)
   P_SYSCR.IRR1.BIT.IRRTA = 0;
   P_IO.PDR9.BIT.P93 ^= 1;
   P_IO.PDR9.BIT.P92 ^= 1;
}
.
.
__interrupt(vect=16) void INT_TimerG(void) {/* sleep(); */}
// vector 17 Reserved

// vector 18 SCI3
// vector 19 ADI
__interrupt(vect=19) void INT_ADI(void) {/* sleep(); */}
// vector 20 Direct Transition
__interrupt(vect=20) void INT_Direct_Transition(void) {/* sleep(); */}

//Insert SCI ISR vector address as 0x0440
#pragma section SCI_ISR
static const unsigned short DATA = 0x0440;
//__interrupt(vect=18) void INT_SCI3(void) {/* sleep(); */}
```

9.5 Kernel Code Listing

9.5.1 Flash Kernel Program

The figure below shows the flow chart for “kernel.c”, followed by its code listing.

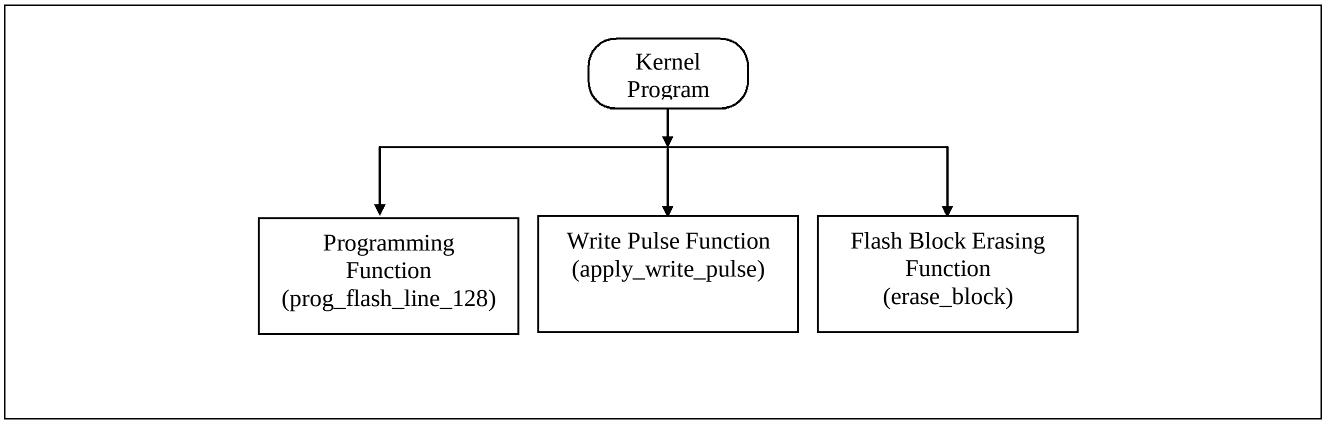

Kernel Program
Programming Function (prog_flash_line_128)
Write Pulse Function (apply_write_pulse)
Flash Block Erasing Function (erase_block)

Figure 31 The Flow Chart for Kernel Program

Note: Please refer to the ‘Flash Memory Programming Mode’ Application note for more details.

```
// Renesas H8/38024F example flash programming and erasing routines
//
// kernel.c
//
// Clock speed = 9.8304MHz
// H8/38024F uses SCI3 for user mode
// Kernel start address - 0xF780


#include "iodefine.h"                    // IO header file
#include <machine.h>

// H8/38024F specific
#define FLASH_SWE P_ROM.FLMCR1.BIT.SWE
#define FLASH_PSU P_ROM.FLMCR1.BIT.PSU
#define FLASH_P   P_ROM.FLMCR1.BIT.P
#define FLASH_PV  P_ROM.FLMCR1.BIT.PV
#define FLASH_EBR1   P_ROM.EBR.BYTE
#define FLASH_ESU P_ROM.FLMCR1.BIT.ESU
#define FLASH_E   P_ROM.FLMCR1.BIT.E
#define FLASH_EV  P_ROM.FLMCR1.BIT.EV
#define  FLASH_FENR  P_ROM.FENR.BIT.FLSHE

// H8/38024F specific
#define MAX_FLASH_ADDR                   0x8000
#define FLASH_LINE_SIZE                  128
#define NO_OF_FLASH_BLOCKS               5
#define XTAL                             9830400L
#define MAX_PROG_COUNT                   1000
#define MAX_ERASE_ATTEMPTS               100
#define BLANK_VALUE                      0xFFFF      // 0xFFFFFFFF for SH,
                                                     //0xFFFF for H8S/300H

// array below should contain the start addresses of the flash memory blocks
// final array element should contain the end address of the flash memory (+1)

#pragma section kernel_const //only applicable for M2_init_ws
                             //additional constant section define needed

const unsigned long eb_block_addr [NO_OF_FLASH_BLOCKS + 1] =  {
   0x00000000L,
   0x00000400L,
   0x00000800L,
   0x00000C00L,
   0x00001000L,
   0x00008000L    /* max flash address + 1 */
};

#define BLANK                            1
#define NOT_BLANK                        2
#define PROG_PASS                        0x01
#define PROG_FAIL                        0x02
#define ERASE_PASS                       0x01
#define ERASE_FAIL                       0x02
```

```
// delay values
// note this is xtal frequency specific
// these values are for the H8/38024F Timer F with a clock divider of 4
#define ONE_USEC                        ((1L * XTAL) / 8000000L)
#define TWO_USEC                        ((2L * XTAL) / 8000000L)
#define FOUR_USEC                       ((4L * XTAL) / 8000000L)
#define FIVE_USEC                       ((5L * XTAL) / 8000000L)
#define TEN_USEC                        ((1L * XTAL) / 800000L)
#define TWENTY_USEC                     ((2L * XTAL) / 800000L)
#define THIRTY_USEC                     ((3L * XTAL) / 800000L)
#define FIFTY_USEC                      ((5L * XTAL) / 800000L)
#define ONE_HUNDRED_USEC                ((1L * XTAL) / 80000L)
#define TWO_HUNDRED_USEC                ((2L * XTAL) / 80000L)
#define TEN_MSEC                        ((1L * XTAL) / 800L)

// typedef for reading the flash memory
// should be the size of the data bus connection to the flash memory
typedef unsigned short read_datum;

// function prototypes
unsigned char prog_flash_line_128 (unsigned long t_address, union
char_rd_datum_union *p_data);
void delay (unsigned short);
void init_delay_timer (void);
unsigned char erase_block (unsigned char block_num);
void apply_write_pulse(unsigned short prog_pulse);
extern void sci_put(char byte);
// variables
volatile unsigned long delay_counter;


union char_rd_datum_union {
   unsigned char c[FLASH_LINE_SIZE];
   read_datum u[FLASH_LINE_SIZE / sizeof (read_datum)];
} prog_data;

//DEFINE SECTION FOR KERNEL PROGRAM
#pragma section kernel

/*************************************************************
/*
/*    FUNCTION       : prog_flash_line_128
/*    DESCRIPTION    : program 128 bytes of flash memory
/*    INPUT          : flash start address,
/*                     program data pointer
/*    OUTPUT         : PROG_PASS if programming is successful
/*                     PROG_FAIL if programming is unsucessful
/*    Other information:
/*    t_address is the start address for the flash line to
/*    be programmed and must be on a flash line boundary e.g.
/*    multiple of 128 (this is not checked and so must be
/*    ensured by the caller) data to be programmed should be
/*    passed to this function in the form of a 'char_rd_datum_union'
/*    union pointer data must be written to the flash in byte units
```

```
/*
/*    Please note that for the H8/38024F during the dummy write,
/*    setting the PSU  and P bits no RTS intructions are permitted.
/*    Therefore no functions calls are allowed.
/*
/*    For this reason at these points in this function the code from
/*    the 'delay' function has been inlined to eliminate any RTS
/*    instructions. For further information on this see the Flash ROM
/*    section of the H8/38024F hardware manual version 4 or later.
/*
/*************************************************************/

// Program 128 bytes functions start here
unsigned char prog_flash_line_128 (unsigned long t_address, union
char_rd_datum_union *p_data)
{
   unsigned char i;
   unsigned short n_prog_count;
   // loop counter for programming attempts (0 -> MAX_PROG_COUNT)
   unsigned short d;
   // variable used for various loop counts
   unsigned short ax;
   // loop counter for incrementing 'uc_v_write_address'

   // pointer (an unsigned short produces more efficient code than unsigned
   // char in this case)
   unsigned char m;
   // flag to indicate if re-programming is required (1=yes, 0=no)
   unsigned char *dest_address;          // pointer for writing to flash
   unsigned char *uc_v_write_address;
   // pointer for writing to address to be verified
   read_datum *ul_v_read_address;        // pointer for reading verify address
   union char_rd_datum_union additional_prog_data, re_program_data;
                                         // storage on stack for intermediate
                                         // programming data
   //Init Timer F start
   // 16 bit timer F counter, System clock / 4 selected
   P_TMRF.TCRF.BYTE = 0x86;

   //TCF cleared when TCF and OCRF match
   P_TMRF.TCSRF.BIT.CCLRH = 1;
   //Init Timer F end

   // enable access to the flash registers
   FLASH_FENR = 1;

   // enable flash writes
   FLASH_SWE = 1;

   // wait tSSWE (1 us)
   delay(ONE_USEC);

   // copy data from program data area to reprogram data area
   for (d=0; d<FLASH_LINE_SIZE; d++)
```

```
    {
       re_program_data.c[d] = p_data->c[d];
    }

    // program the data in FLASH_LINE_SIZE (128) byte chunks
    for (n_prog_count=0; n_prog_count<MAX_PROG_COUNT; n_prog_count++)
    {
       // clear reprogram required flag
       m = 0;

       // copy data from reprogram data area into the flash with byte wide
       // access
       dest_address = (unsigned char *) t_address;

       for (d=0; d<FLASH_LINE_SIZE; d++)
       {
          *dest_address++ = re_program_data.c[d];
       }

       // to minimise code space the code to apply a write pulse has been
       // placed into a separate function called 'apply_write_pulse'
       if (n_prog_count < 6)
       {
          apply_write_pulse(THIRTY_USEC);
       }
       else
       {
          apply_write_pulse(TWO_HUNDRED_USEC);
       }

       // verify the data via word wide reads
       uc_v_write_address = (unsigned char *) t_address;
       ul_v_read_address = (read_datum *) t_address;

       // enter program verify mode
       FLASH_PV = 1;

       // wait tSPV (4 us)
       delay (FOUR_USEC);


       // read data in read_datum size chunks
       // verify loop
       for (d=0; d<(FLASH_LINE_SIZE / sizeof(read_datum)); d++)
       {
          // dummy write of H'FF to verify address
          *uc_v_write_address = 0xff;

          // see note at beginning of function
          // no RTS allowed here so 'apply_write_pulse' function inlined

          P_TMRF.OCRF.BYTE.H = (TWO_USEC)>>8;
          P_TMRF.OCRF.BYTE.L = (TWO_USEC);
```

```
      // Clear compare match flag
      P_TMRF.TCSRF.BIT.CMFH = 0;

      // Clear counter and start the timer F
      P_TMRF.TCF.BYTE.H = 0;
      P_TMRF.TCF.BYTE.L = 0;

      // Loop until we have a compare match
      while (P_TMRF.TCSRF.BIT.CMFH == 0);

      // increment this pointer to get to next verify address
      for (ax=0; ax<sizeof(read_datum); ax++)
      uc_v_write_address++;

      // read verify data
      // check with the original data
      if (*ul_v_read_address != p_data->u[d])
      {
         // 1 or more bits failed to program
         //
         // set the reprogram required flag
         m = 1;
      }

      //Enable watchdog timer
      P_WDT.TCSRW.BYTE = 0x5A;
      P_WDT.TCW = 0x00;
      P_WDT.TCSRW.BYTE = 0xF4;

      // check if we need to calculate additional programming data
      if (n_prog_count < 6)
      {
         // calculate additional programming data
         // simple ORing of the reprog and verify data
         additional_prog_data.u[d] = re_program_data.u[d] |
         *ul_v_read_address;
      }

      // calculate reprog data
      re_program_data.u[d] = p_data->u[d] | ~(p_data->u[d] |
      *ul_v_read_address);

      // increment the verify read pointer
      ul_v_read_address++;

      //Disable watchdog timer
      P_WDT.TCSRW.BYTE = 0xF2;
   } // end of verify loop
   // exit program verify mode
   FLASH_PV = 0;

   // check if additional programming is required
   if (n_prog_count < 6)
```

```
      {
         // perform additional programming
         //
         // copy data from additional programming area to flash memory
         dest_address = (unsigned char *) t_address;
         for (d=0; d<FLASH_LINE_SIZE; d++)
         {
            *dest_address++ = additional_prog_data.c[d];
         }

         apply_write_pulse(TEN_USEC);
      }
      // check if flash line has successfully been programmed
      if (m == 0)
      {
         // program verified ok
         //
         // disable flash writes
         FLASH_SWE = 0;

         // wait tCSWE (100 us)
         delay (ONE_HUNDRED_USEC);

         // end of successful programming
         // disable access to the flash registers
         FLASH_FENR = 0;
         return (PROG_PASS);
      }

   }  // end of for loop (n<MAX_PROG_COUNT) at this point we have made
      // MAX_PROG_COUNT programming attempts

   // failed to program after MAX_PROG_COUNT attempts
   // disable flash writes
   FLASH_SWE = 0;

   // wait tCSWE (100 us)
   delay (ONE_HUNDRED_USEC);

   // end of failed programming
   // disable access to the flash registers
   FLASH_FENR = 0;
   return (PROG_FAIL);
}
// Program 128 bytes functions end here
```

```
/************************************************************
/*
/*  FUNCTION    :apply_write_pulse
/*  DESCRIPTION :Applies programming pulse to flash memory
/*  INPUT      :prog_pulse = 30us, 200us or 10us
/*  OUTPUT     :None
/*************************************************************/
// apply_write_pulse functions start here
void apply_write_pulse(unsigned short prog_pulse)
{

   //Enable watchdog timer
   P_WDT.TCSRW.BYTE = 0x5A;
   P_WDT.TCW = 0x00;
   P_WDT.TCSRW.BYTE = 0xF4;

   // enter program setup mode
   FLASH_PSU = 1;

   // no RTS allowed here so 'apply_write_pulse' function inlined

   P_TMRF.OCRF.BYTE.H = FIFTY_USEC>>8;
   P_TMRF.OCRF.BYTE.L = FIFTY_USEC;

   // Clear compare match flag
   P_TMRF.TCSRF.BIT.CMFH = 0;

   // Clear counter and start the timer F
   P_TMRF.TCF.BYTE.H = 0;
   P_TMRF.TCF.BYTE.L = 0;

   // Loop until we have a compare match
   while (P_TMRF.TCSRF.BIT.CMFH == 0);


   // start programming pulse
   FLASH_P = 1;

   // no RTS allowed here so 'apply_write_pulse' function inlined

   P_TMRF.OCRF.BYTE.H = prog_pulse>>8;
   P_TMRF.OCRF.BYTE.L = prog_pulse;

   // Clear compare match flag
   P_TMRF.TCSRF.BIT.CMFH = 0;

   // Clear counter and start the timer F
   P_TMRF.TCF.BYTE.H = 0;
   P_TMRF.TCF.BYTE.L = 0;

   // Loop until we have a compare match
   while (P_TMRF.TCSRF.BIT.CMFH == 0);

   // stop programming
```

```
    FLASH_P = 0;

    // delay (FIVE_USEC);
    P_TMRF.OCRF.BYTE.H = FIVE_USEC>>8;
    P_TMRF.OCRF.BYTE.L = FIVE_USEC;

    // Clear compare match flag
    P_TMRF.TCSRF.BIT.CMFH = 0;

    // Clear counter and start the timer F
    //P_TMRF.TCF.WORD = 0;
    P_TMRF.TCF.BYTE.H = 0;
    P_TMRF.TCF.BYTE.L = 0;

    // Loop until we have a compare match
    while (P_TMRF.TCSRF.BIT.CMFH == 0);


    // exit program setup mode
    FLASH_PSU = 0;

    // wait tCPSU (5 us)
    // delay (FIVE_USEC);
    P_TMRF.OCRF.BYTE.H = FIVE_USEC>>8;
    P_TMRF.OCRF.BYTE.L = FIVE_USEC;

    // Clear compare match flag
    P_TMRF.TCSRF.BIT.CMFH = 0;

    // Clear counter and start the timer F
    //P_TMRF.TCF.WORD = 0;
    P_TMRF.TCF.BYTE.H = 0;
    P_TMRF.TCF.BYTE.L = 0;

    // Loop until we have a compare match
    while (P_TMRF.TCSRF.BIT.CMFH == 0);
    //Disable watchdog timer
    P_WDT.TCSRW.BYTE = 0xF2;
}
// apply_write_pulse functions end here
```

```
/************************************************************
/*
/*  FUNCTION    :erase_block
/*  DESCRIPTION :Erase flash memory block
/*  INPUT      :block_num = 0,1,2,3,4
/*  OUTPUT     :ERASE_PASS is attempt is successful
/*           ERASE_FAIL is attempt fails
/*************************************************************/
// erase block functions start here
unsigned char erase_block (unsigned char block_num)
{
   unsigned char erase, ax, x;
   unsigned long attempts;
   read_datum *ul_v_read;
   unsigned char *uc_v_write;

   //Init Timer F start
   // 16 bit timer F counter, System clock / 4 selected
   P_TMRF.TCRF.BYTE = 0x86;

   //TCF cleared when TCF and OCRF match
   P_TMRF.TCSRF.BIT.CCLRH = 1;

   // check that block is not already erased
   erase = BLANK;
   for (attempts=eb_block_addr[block_num]; attempts<eb_block_addr[block_num +
   1]; attempts++)
   {
      if ( *(unsigned char *) attempts != 0xff)
         erase = NOT_BLANK;
   }

   if (erase == BLANK)
      return ERASE_PASS;
   else
   {
      // block needs erasing
      //
      // enable access to the flash registers
      FLASH_FENR = 1;

      // enable flash writes
      FLASH_SWE = 1;

      // wait tSSWE (1us)
      delay (ONE_USEC);

      // initialise the attempts counter
      // 0 as we check for less than MAX (not <= MAX)
      attempts = 0;

      // set the correct EB bit in correct EBR register
      FLASH_EBR1 = 1<<block_num;
      erase = 0;
```

```
    while ( (attempts < MAX_ERASE_ATTEMPTS) && (erase == 0) )
    {
       // increment the attempts counter

       attempts++;
       // enter erase setup mode
       FLASH_ESU = 1;

       // wait tSESU (100 us)
       delay (ONE_HUNDRED_USEC);

       // start erasing
       FLASH_E = 1;

       // wait tSE (10 ms)
       delay (TEN_MSEC);

       // stop erasing
       FLASH_E = 0;

       // wait tCE (10 us)
       delay (TEN_USEC);

       // exit erase setup mode
       FLASH_ESU = 0;

       // wait tCESU (10 us)
       delay (TEN_USEC);

       // enter erase verify mode
       FLASH_EV = 1;

       // wait tSEV (20 us)
       delay (TWENTY_USEC);

       // verify flash has been erased
       // setup the pointers for reading and writing the flash
       ul_v_read = (read_datum *) eb_block_addr [block_num];
       uc_v_write = (unsigned char *) eb_block_addr [block_num];

       erase = 1;
       while ( (erase == 1) && ( ul_v_read < (read_datum *) eb_block_addr
       [block_num + 1] ) )
       {
          // this loop will exit either when one word is not erased ('erase'
          // becomes 0)
          // or all addresses have been read as erased ('erase' stays as 1)
          // if 'erase' stays as 1 the outer while loop will exit as the
          // block has been erased
          //
          // dummy write
          *uc_v_write = 0xff;

          // see note at beginning of function
```

```
            // no RTS allowed here so 'apply_write_pulse' function inlined
            P_TMRF.OCRF.BYTE.H = TWO_USEC>>8;
            P_TMRF.OCRF.BYTE.L = TWO_USEC;

            // Clear compare match flag
            P_TMRF.TCSRF.BIT.CMFH = 0;

            // Clear counter and start the timer F
            P_TMRF.TCF.BYTE.H = 0;
            P_TMRF.TCF.BYTE.L = 0;


            // Loop until we have a compare match
            while (P_TMRF.TCSRF.BIT.CMFH == 0);


            if (*ul_v_read != BLANK_VALUE)
            {
               // this word is not erased yet
               erase = 0;
            }
            else
            {
               // advance to the next byte write address
               for (ax=0; ax<sizeof(read_datum); ax++)
                  uc_v_write++;

               // advance to the next verify read address
               ul_v_read++;
            }
         }

         // exit erase verify mode

         FLASH_EV = 0;

         // wait tCEV (4 us)
         delay (FOUR_USEC);
      }  // end of outer while loop

      // end either of erase attempts or block has been erased ok
      //
      // disable flash writes
      FLASH_SWE = 0;

      // wait tCSWE (100 us)
      delay (ONE_HUNDRED_USEC);

      // check if block has been erased ok
      if (erase == 1)
      {
         // successfully erased
         // disable access to the flash registers
         FLASH_FENR = 0;
         return ERASE_PASS;
      }
      else
      {
         // failed to erase this block
         // disable access to the flash registers
         FLASH_FENR = 0;
         return ERASE_FAIL;
      }
   }
}
// erase block functions end here

/*************************************************************
/*
/*  FUNCTION    :delay
/*  DESCRIPTION :Timer F delay function
/*  INPUT      :d = time in us
/*  OUTPUT     :None
/*************************************************************/
// delay functions start here
void delay (unsigned short d)
{
   // load compare match value into the output compare register

   P_TMRF.OCRF.BYTE.H = d>>8;
   P_TMRF.OCRF.BYTE.L = d;

   // Clear compare match flag
   P_TMRF.TCSRF.BIT.CMFH = 0;

   // Clear counter and start the timer F
   P_TMRF.TCF.BYTE.H = 0;
   P_TMRF.TCF.BYTE.L = 0;

   // Loop until we have a compare match
   while (P_TMRF.TCSRF.BIT.CMFH == 0);

   P_TMRF.TCSRF.BIT.CMFH = 0;
}
// delay functions start here

#pragma section
//end of kernel section
```

9.5.2 ROM to RAM Mapping Program

The following code listing is the ROM to RAM mapping section declaration of "ROMtoRAM.c".

This code, which is stored in ROM but executed in RAM, has to be treated differently. The section has to be correctly mapped, to allow the compiler to generate the correct executing code.

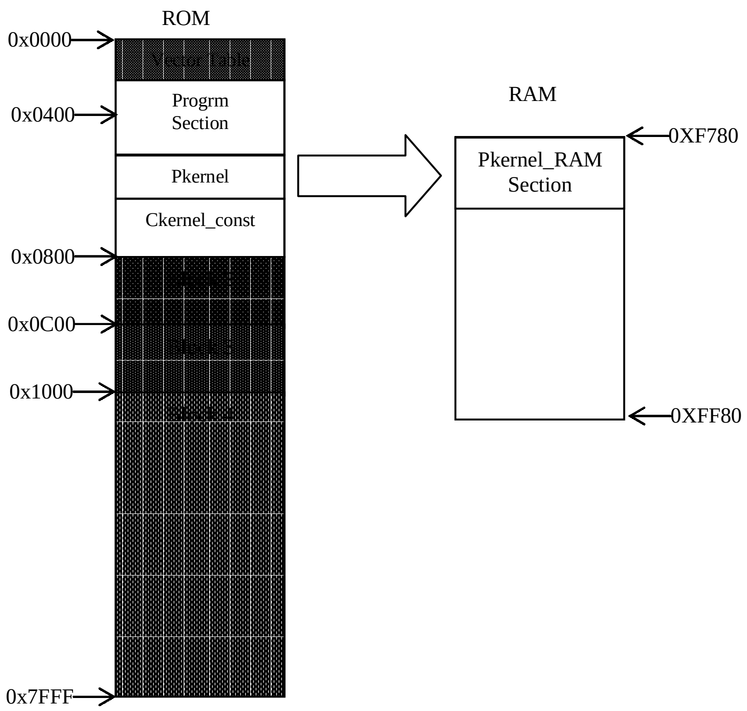
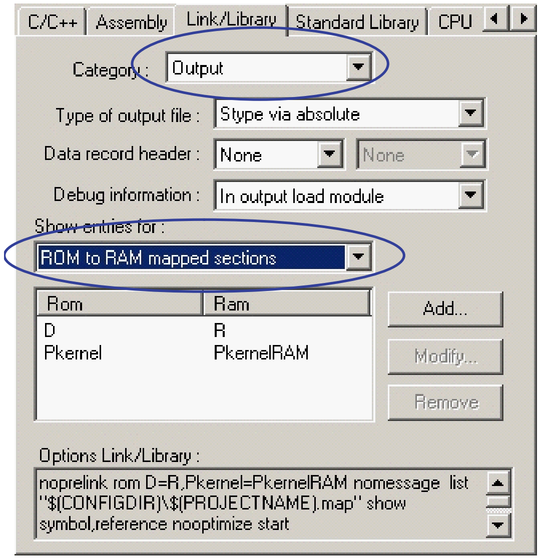


Figure 32 Memory Map for Kernel Section



Category : Output

Type of output file : Stype via absolute

Data record header : None None

Debug information : In output load module

Show entries for : ROM to RAM mapped sections

Rom	Ram
D	R
Pkernel	PkernelRAM

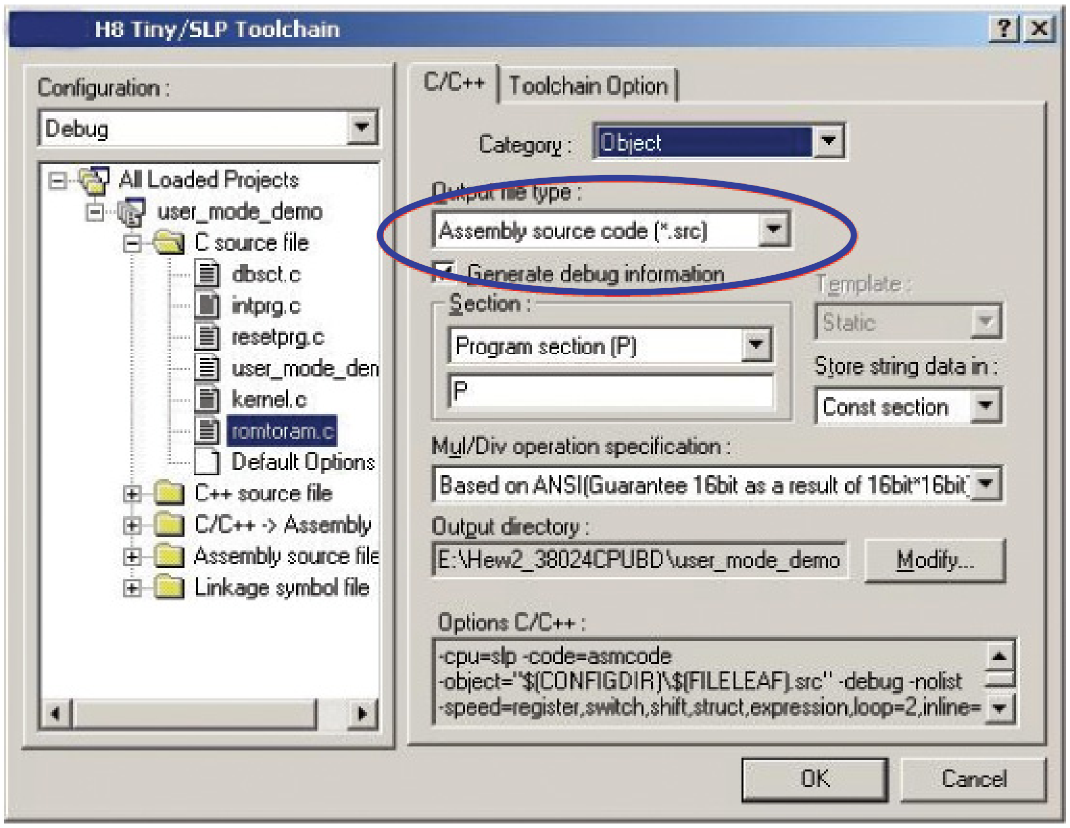
Options Link/Library :

```
noprelink rom D=R,Pkernel=PkernelRAM nomessage list
"$(CONFIGDIR)\$(PROJECTNAME).map" show
symbol,reference nooptimize start
```

Figure 33 ROM to RAM Section Mapping Configuration

```
#pragma asm
   .SECTION PkernelRAM,CODE,ALIGN=2

   .SECTION Pkernel,CODE,ALIGN=2

   .SECTION Ckernel_const,DATA,ALIGN=2

;Start Address of Section ROMCODE - kernel
__PkernelBegin    .DATA.W (STARTOF Pkernel)

;End Address of Section ROMCODE - kernel
__PkernelEnd .DATA.W (STARTOF Pkernel) + (SIZEOF Pkernel)

;Start Address of Section RAMCODE - kernel
__Pkernel_RAMBegin .DATA.W (STARTOF PkernelRAM)

   .EXPORT __PkernelBegin
   .EXPORT __PkernelEnd
   .EXPORT __Pkernel_RAMBegin
#pragma endasm
```

Note: The above code is written in assembly languages. Thus "Assembly source code (*.src)" output file type needs to be configured from the *Renesas H8 Tiny/SLP Toolchain* in the *Options* menu as below:



Category : Object

Output file type : Assembly source code (*.src)

Generate debug information

Section : Program section (P)

P

Store string data in : Const section

Mul/Div operation specification : Based on ANSI(Guarantee 16bit as a result of 16bit*16bit)

Output directory : E:\Hew2_38024CPUBD\user_mode_demo

Options C/C++ :

```
-cpu=slp -code=asmcode
-object="$(CONFIGDIR)\$(FILELEAF).src" -debug -nolist
-speed=register,switch,shift,struct,expression,loop=2,inline=
```

Figure 34 ROM to RAM .c File Configuration

10. Serial Communication Debugging Technique

If modification is made to the interfacing protocol, programmer can make use of the following technique to assist him/her in troubleshooting. A simple serial communication tool can be built to monitor the TX & RX lines between the PC and the SLP.

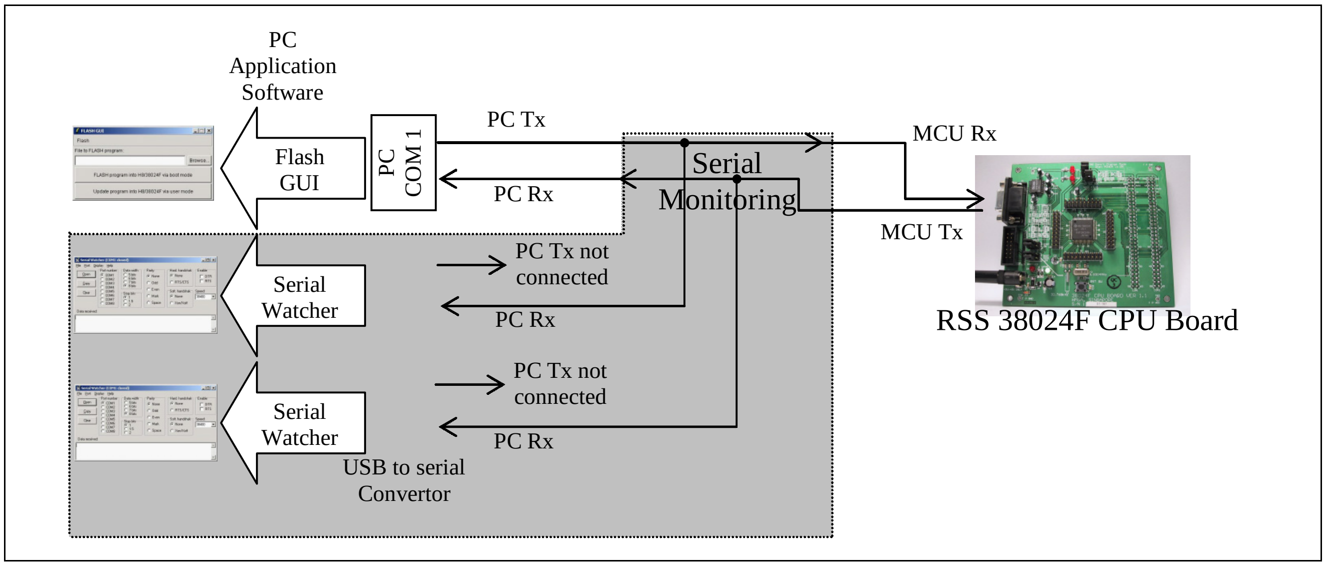


Figure 35 Serial Communication Monitoring Tool

In this case the PC will require three serial ports:

1. For the Flash GUI to control the SLP
2. To monitor the PC TX line
3. To monitor the PC RX line

A good software for monitoring COM port activity is the *"SerialWatcher.exe"*. It is able to display data in hexadecimal and ASCII and is able to support up to 8 COM ports at a time.

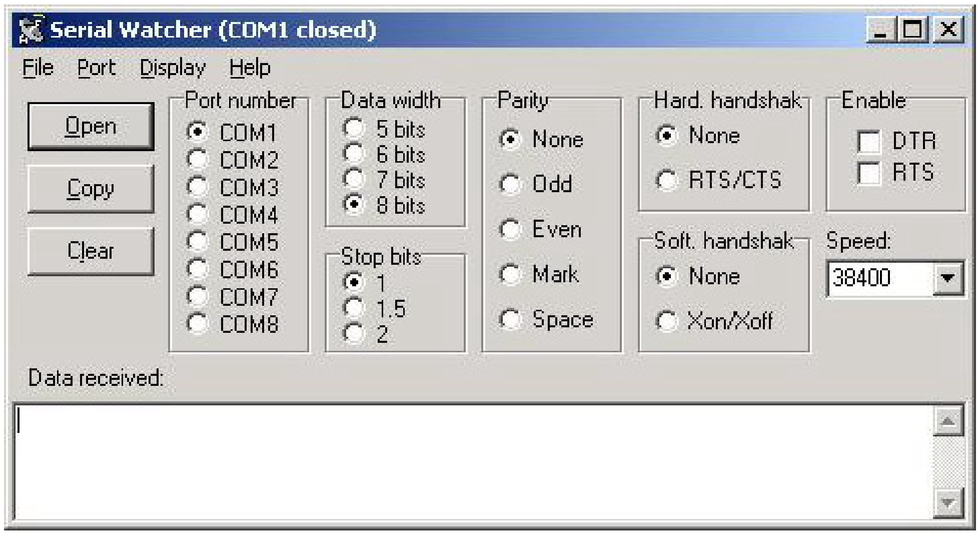

Serial Watcher (COM1 closed)
File Port Display Help
Open
Copy
Clear
Port number: [✓ COM1] COM2 COM3 COM4 COM5 COM6 COM7 COM8
Data width: 5 bits 6 bits 7 bits [✓ 8 bits]
Stop bits: [✓ 1] 1.5 2
Parity: [✓ None] Odd Even Mark Space
Hard. handshak: [✓ None] RTS/CTS
Soft. handshak: [✓ None] Xon/Xoff
Enable: DTR RTS
Speed: 38400
Data received:

Figure 36 Serial Watcher 2.0.4 for Windows

The user may download the serial watcher software from http://www.pcremotecontrol.com/serialwatcher.zip .

11. References

Tcl Related:

1. http://www.activestate.com/Products/ActiveTcl/
2. http://freewrap.sourceforge.net/

Other Related Application Notes:

1. F-ZTAT^{TM} Microcomputer On-Board Programming (Application Note ref. no: ADE-502-042, http://renesas.com,)
2. F-ZTAT^{TM} Microcomputer Single Power Supply F-ZTAT^{TM} On-Board Programming, (Application Note ref. no: ADE-502-055, http://renesas.com)

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Sep.10.04	—	First edition issued