

To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: <http://www.renesas.com>

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (<http://www.renesas.com>)

Send any inquiries to <http://www.renesas.com/inquiry>.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
 - “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
 - “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
 - “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

H8 Family, H8S Family, SuperH RISC Engine Family

Flash Memory Programming Routines

Introduction

All Renesas microcontrollers featuring Flash memory have the ability to self program their Flash memory. This opens up the opportunity to explore new applications and enhance existing ones. For example firmware can be updated in the field via a modem, Internet, wireless etc or motor characterisation data can be changed throughout its lifetime.

At the time of writing Renesas are manufacturing Flash microcontrollers with a 0.18 μ m process complementing the 0.6 μ m and 0.35 μ m based devices. The objective of this apps note is to give an overview of programming and erasing 0.6 μ m, 0.35 μ m and 0.18 μ m based 16-bit and 32-bit microcontrollers and to provide example routines for doing this written in 'C'.

Contents

FLASH MEMORY PROGRAMMING MODES	3
0.6 μ M ALGORITHMS	4
0.6 μ M PROGRAM/PROGRAM-VERIFY	5
0.6 μ M ERASE/ERASE-VERIFY	8
0.35 μ M ALGORITHMS	12
0.35 μ M PROGRAM/PROGRAM-VERIFY	12
0.35 μ M ERASE/ERASE-VERIFY	18
*IMPORTANT NOTE RELATING TO 0.35 μ M DEVICES	20
0.18 μ M ALGORITHMS	21
0.18 μ M PROGRAMMING	21
0.18 μ M ERASING	27
SUMMARY	29
APPENDIX A – RENESAS 0.6 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR H8S/2144F	30
APPENDIX B – RENESAS 0.6 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR SH7045F	41
APPENDIX C – RENESAS 0.35 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR H8S/2612F	51
APPENDIX D – RENESAS 0.35 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR SH7047F	61
APPENDIX E – RENESAS 0.35 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR H8/3664F MICROCONTROLLER.....	71
APPENDIX F – RENESAS 0.18 μ M FLASH PROGRAMING & ERASING ROUTINES FOR H8/3069F.....	81
APPENDIX G – RENESAS 0.18 μ M FLASH PROGRAMING & ERASING ROUTINES FOR SH7058F	94
WEBSITE AND SUPPORT	106

Flash Memory Programming Modes

Renesas Flash microcontrollers typically have three programming modes, PROM, boot and user.

PROM mode requires the use of an external 'EPROM' type programmer where the microcontroller is placed into a socket and programmed. This method offers high programming speeds but lacks flexibility and has limited use in the field.

Boot mode is entered by setting values on a combination of the micro's pins and resetting the device. The micro will then execute a 'hidden' program which erases the Flash memory for security purposes, auto-bauds with a host and then allows a programming kernel to be downloaded into the internal RAM of the micro and executed. This mode allows unprogrammed devices to have their Flash memory programmed in-circuit and in the field. This mode is supported by PC hosted applications such as FDT (Flash Development Toolkit) available from <http://www.renesas.com> It should be noted that in this mode the Flash memory is erased and so must be completely re-programmed each time it is used and that the SCI port used in the boot process is fixed.

The 0.18 μ m devices introduce an additional boot mode called 'user boot mode'. In this mode the device boots from an additional area of Flash, typically 8kB in size and starting from address 0, which takes the place of the 'normal' user Flash memory. What differentiates user boot mode from boot mode is that this additional area of Flash can be programmed by the user making the implementation of a bootloader a simpler prospect. It should be noted that the user boot mode area of Flash can only be programmed from 'normal' boot mode. When in user boot mode the 'normal' user Flash area can be erased and programmed. During the erase sequence of 'normal' boot mode both the 'normal' user area of Flash and the user boot mode area of Flash are erased.

User mode offers the most flexible approach to in-field programming. With this mode the micro is able to reprogram itself by copying the required routines from existing memory contents into RAM or external memory and running from there. This method allows partial erasing and reprogramming of the memory and is particularly suited to bootloader type scenarios. Unlike boot mode the way data is supplied to the device is not limited to a particular SCI channel as, by its very nature, it is user defined and so can be via a parallel interface, wireless link or across the Internet etc.

In all cases it is important to note that while the Flash memory of the micro is being erased or programmed the Flash memory must not be accessed. This means that the erasing and programming code must run from internal RAM or external memory and interrupts should be disabled (unless in the case of SH the vector table is located to non-Flash memory and the VBR changed accordingly).

It is the intention of this apps note to present 0.6 μ m, 0.35 μ m and 0.18 μ m programming and erasing routines for H8/300H, H8S and SH-2 Renesas microcontroller families that can be used in user mode applications. This apps note will not be concerned with the mechanics of getting data into the device as this will be application specific. As previously mentioned, user mode typically runs routines from internal RAM that have been copied from Flash memory which means that these routines must be linked for RAM but relocated and stored in Flash. There are various methods of achieving this storage and relocation some of which have been covered in other Renesas application notes. Therefore, the reading of application note REG05B0021-0100 is recommended.

All H8S and H8/300H code examples have been developed using HEW (High_Performance Embedded Workshop) v1.3 and Renesas C/C++ compiler version v4.0a. The SH-2 examples have been tested using HEW v1.3 and Renesas C/C++ compiler v6.0a.

0.6 μ m Algorithms

The 0.6 μ m Renesas microcontroller Flash memory has the following characteristics.

The Flash memory must be programmed in units of 32 bytes starting on a 32 byte boundary.

The Flash memory is split into sectors of varying sizes.

Erasing is performed on a sector by sector basis.

The erased state is all 1's.

Programming must be performed in the erased state.

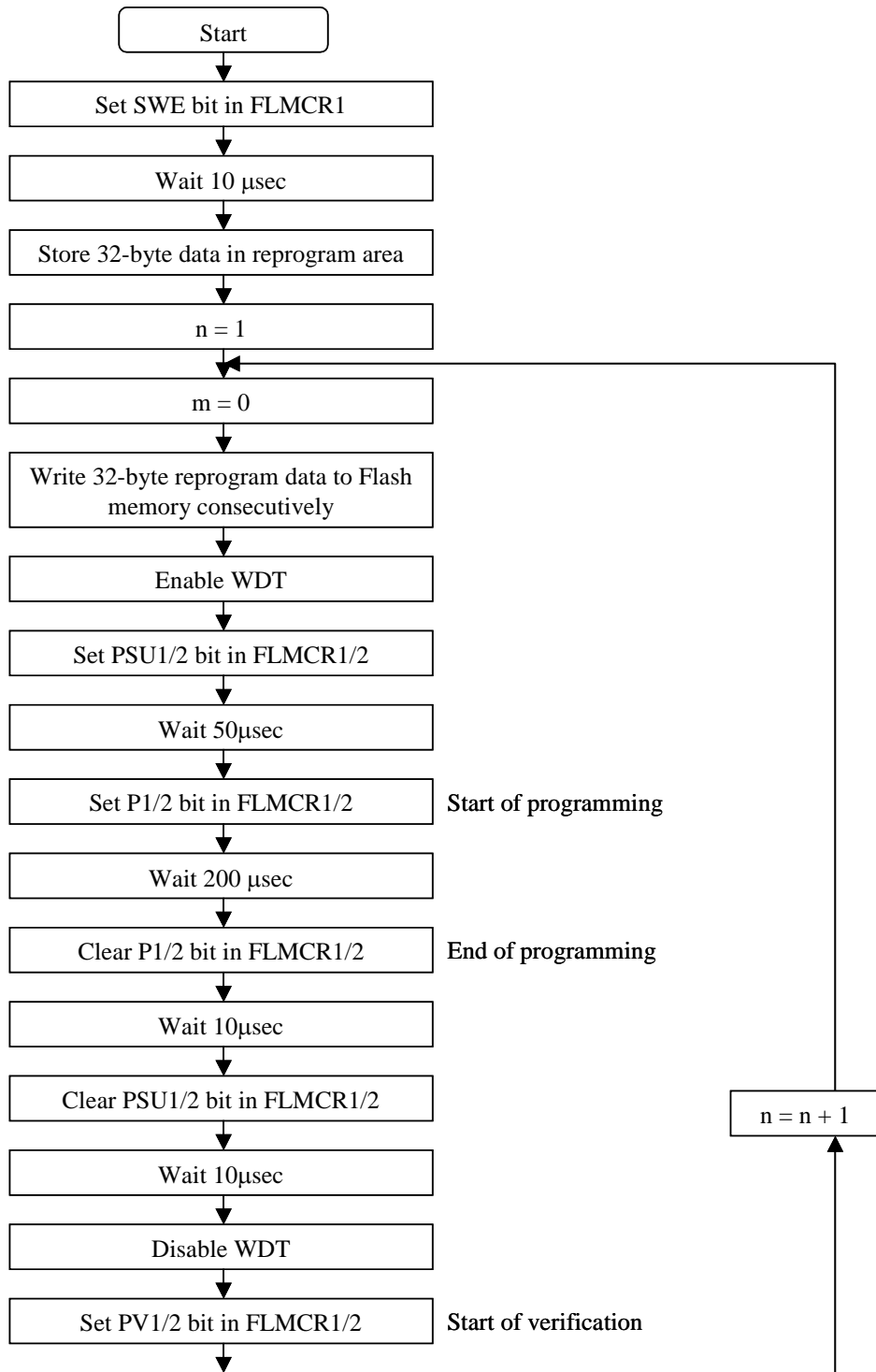
Programming data is written in 16-bit units for H8(S)(300H) and 32-bits for SH-2.

Programming and erase verification data is read in 16-bit units for H8S & H8/300H and 32-bits for SH-2.

Although all 0.6 μ m Renesas Flash microcontrollers essentially have common programming and erasing algorithms it is important that this apps note is read in conjunction with the hardware manual for the device being programmed as there can be subtle differences.

0.6μm Program/Program-Verify

Figure 1 shows the typical program/program verify algorithm for 0.6μm Renesas Flash microcontrollers.



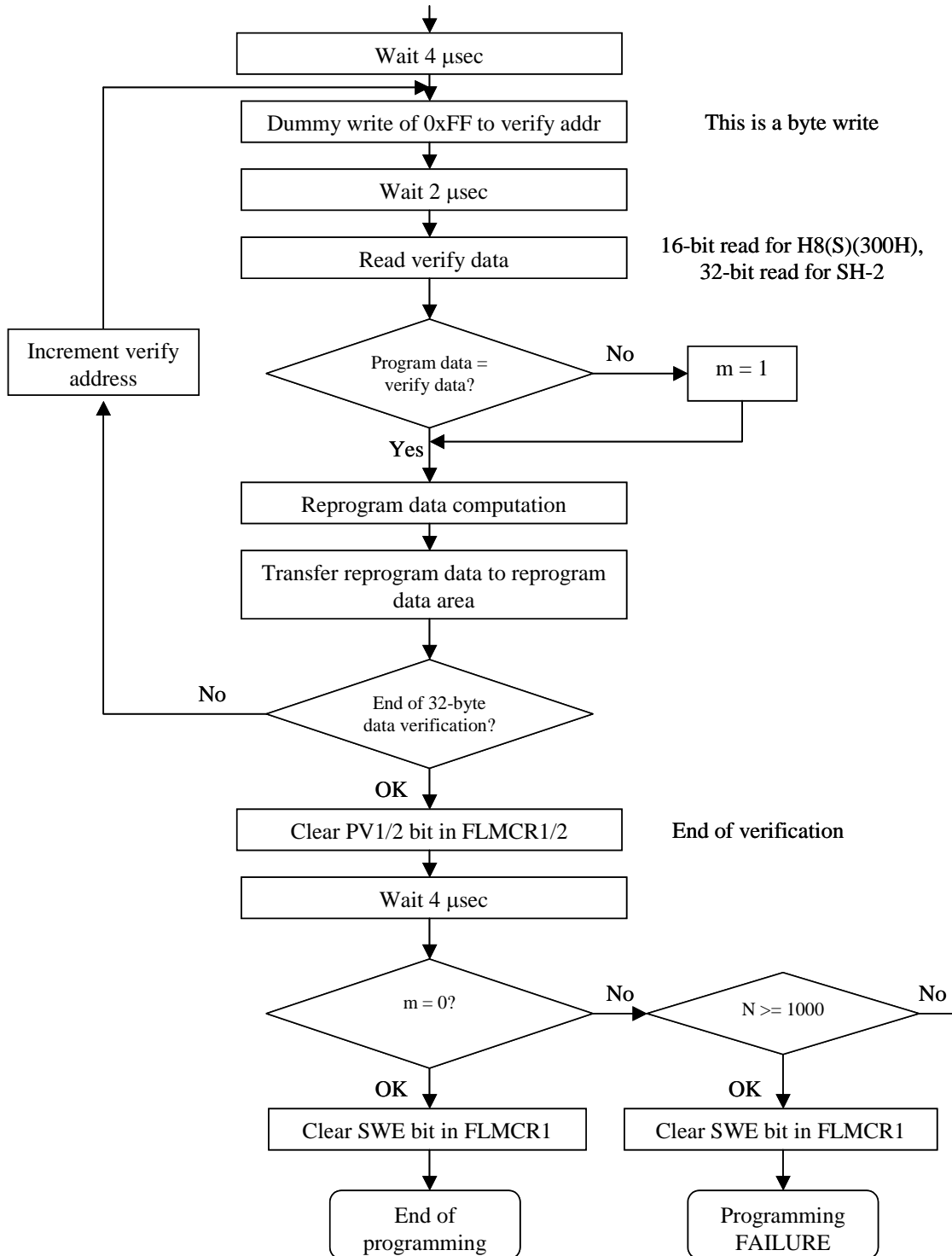


Figure1: 0.6µm Program/Program-Verify Algorithm

Important aspects of the 0.6µm program/program-verify algorithm to note include:

- All delay times are minimum times required to allow the internal signals to settle with one major exception – the time the programming signal (P bit in FLMCR) is set. This time is a MAXIMUM and should not be exceeded.
- Loop counts are maximum values and should not be exceeded.
- When performing a dummy write during the verify stage the dummy write should be performed as a byte access.
- During the verify stage the data read back from the Flash should be compared against the actual data to be programmed and not the reprogram data.
- Programming should only be performed with the Flash cells in the erased ('1') state.

The program/program-verify process is a two stage affair. First an attempt to program a Flash line of 32-bytes is made. Then the Flash memory is put into program-verify mode and the programmed data read back using a 'weak' read of the cells. Here if the data is read back correctly with a 'weak' read then the cell's contents can be guaranteed over the data retention lifetime and temperature range specified for the individual device. If any of the bits fail to stick then reprogram data is calculated that only attempts to reprogram the bits that need programming next time and so avoiding the over-programming of cells that stick early in the programming process. This is repeated until either the Flash memory is programmed successfully or the maximum number of programming attempts is reached.

The reprogram data is calculated according to the following truth table.

Required Data	Verify Data Read	Reprogram Data
0	0	1
0	1	0
1	0	1
1	1	1

Table 1: Reprogramming Data

Appendix A contains source code for implementing the program/program-verify algorithm described above for the H8S and H8/300H. This code has been tested on an H8S/2144F device. This code should be taken as an example and should be modified where necessary for the particular device being programmed and its xtal frequency. It is worth noting that the delays are implemented using a hardware timer and that for the shorter periods the waiting time will be slightly greater than the desired value. This is acceptable as these shorter delays are provided to allow internal signals to settle and so are, as previously mentioned, minimum values.

Appendix B contains example source code for the 0.6 µm program/program verify algorithm for the SH-2 series of Renesas microcontrollers. This code has been tested on an SH7045F device.

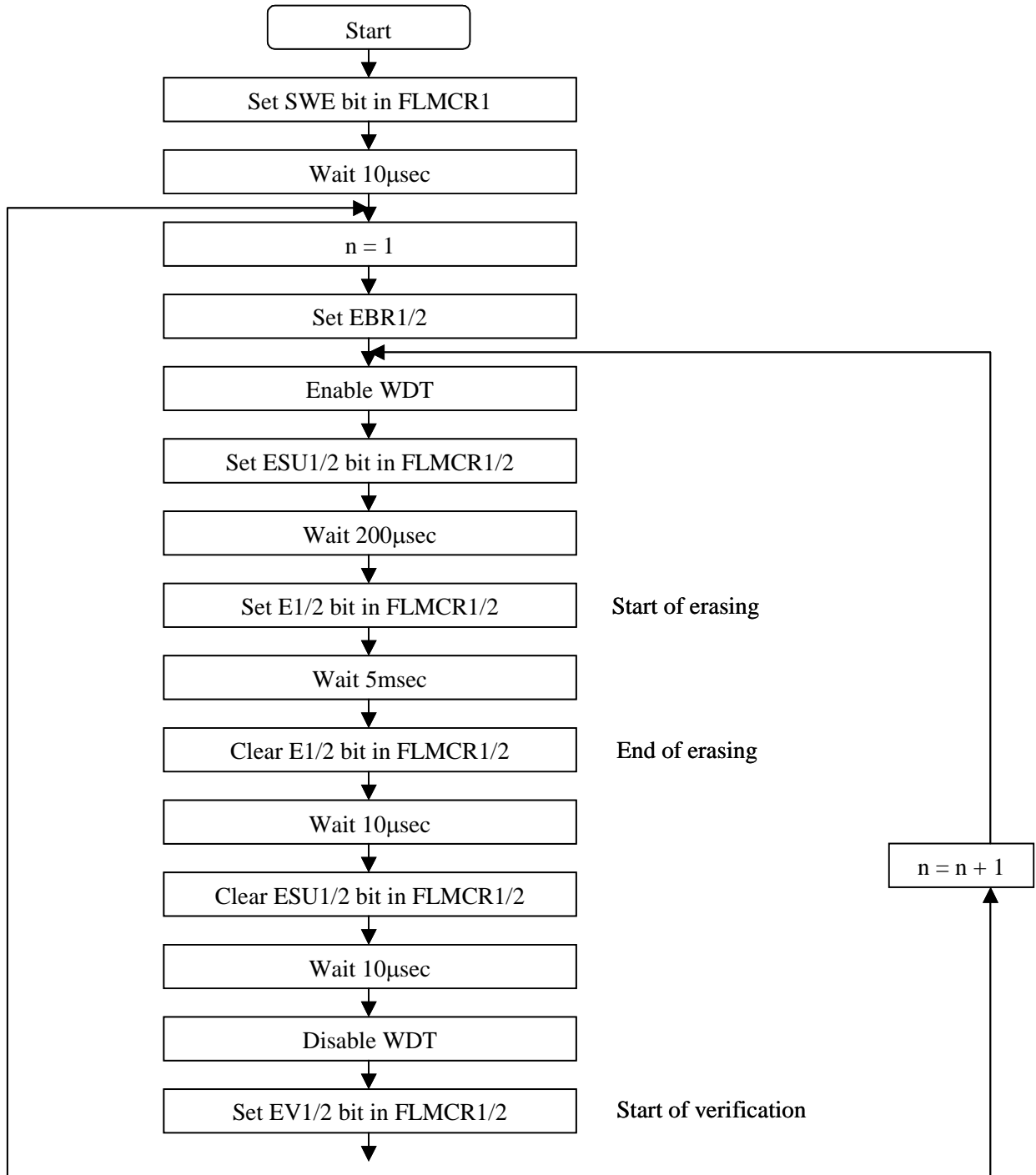
A 32-byte Flash line can be programmed by calling the function 'prog_flash_line_32' which has the following prototype.

```
unsigned char prog_flash_line_32 (unsigned long t_address, union  
char_rd_datum_union *p_data)
```

As can be seen the function is passed two variables. The first, 't_address' is the address of the first byte to be programmed in the Flash memory and must be on a 32-byte boundary. The second variable, 'p_data', is a pointer to a 'char_rd_datum_union' which contains the 32 bytes of data to be programmed into the Flash. The function returns a programming success or failure status byte. This function is identical in the two listings with its functionality being modified by the typedef 'read_datum' which is 16-bits in size for the H8S implementation and 32-bits for the SH-2.

0.6 μ m Erase/Erase-Verify

Figure 2 below shows the typical erase/erase verify algorithm for 0.6 μ m Renesas Flash microcontrollers.



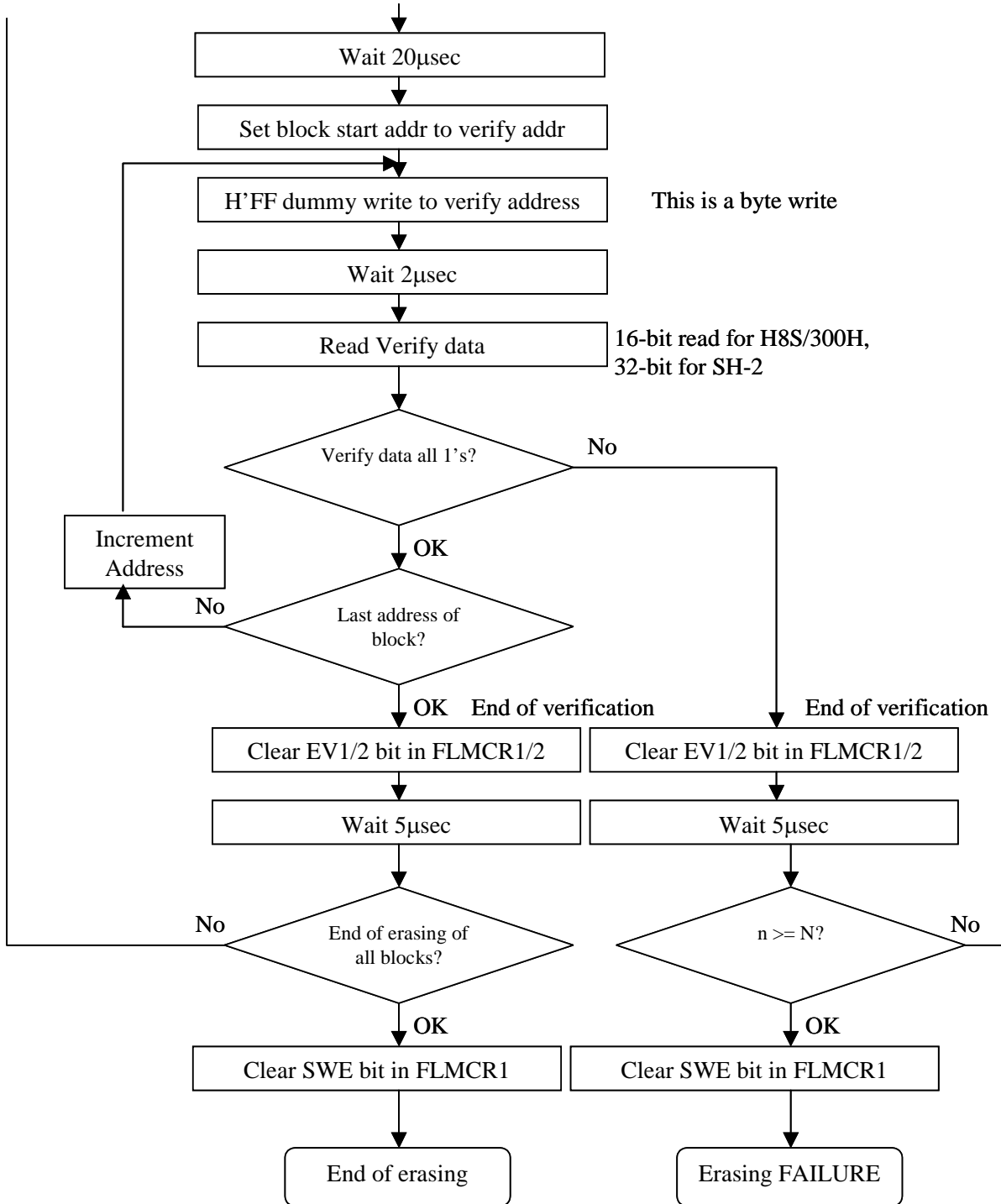


Figure 2: 0.6µm Erase/Erase-Verify Algorithm

Important aspects of the 0.6 μ m erase/erase-verify algorithm to note include:

- All delay times are minimum times required to allow the internal signals to settle with one major exception – the time the erase signal (E bit in FLMCR) is set. This time is a MAXIMUM and should not be exceeded.
- The erase pulse time is in units of msec and the settling times in units of μ sec.
- Loop counts are maximum values and should not be exceeded.
- The dummy write performed during the erase-verify stage should be a byte wide access.
- During the verify stage the Flash should be accessed as 16-bits for H8S/300H and 32-bits for SH-2.
- The erased state is all 1's.
- Pre-programming the Flash contents to '0' is not necessary.
- Only one bit in the EBR registers should be set at any one time as each Flash block must be erased separately.

As with the programming of the Flash memory the erase/erase-verify is a two stage process. An attempt is made to erase the Flash block then the memory is placed into erase verify mode and a 'weak' read of its contents made. If any bit in the Flash block is not set to '1' when read then another attempt is made to erase the block. This process is repeated until either the Flash block is successfully erased or the maximum number of erase attempts is reached.

The source code listings in Appendices A and B contain a function to erase a specified Flash block. The prototype for this function is shown below.

```
unsigned char erase_block_06_um (unsigned char block_num)
```

The function should be passed the number of the Flash block to be erased with the first block being numbered '0'. A success or failure status byte is returned to the caller. The same function can be used with both H8S and SH-2 based 0.6 μ m Flash memory so long as the typedef 'read_datum' is declared accordingly.

0.35 μ m Algorithms

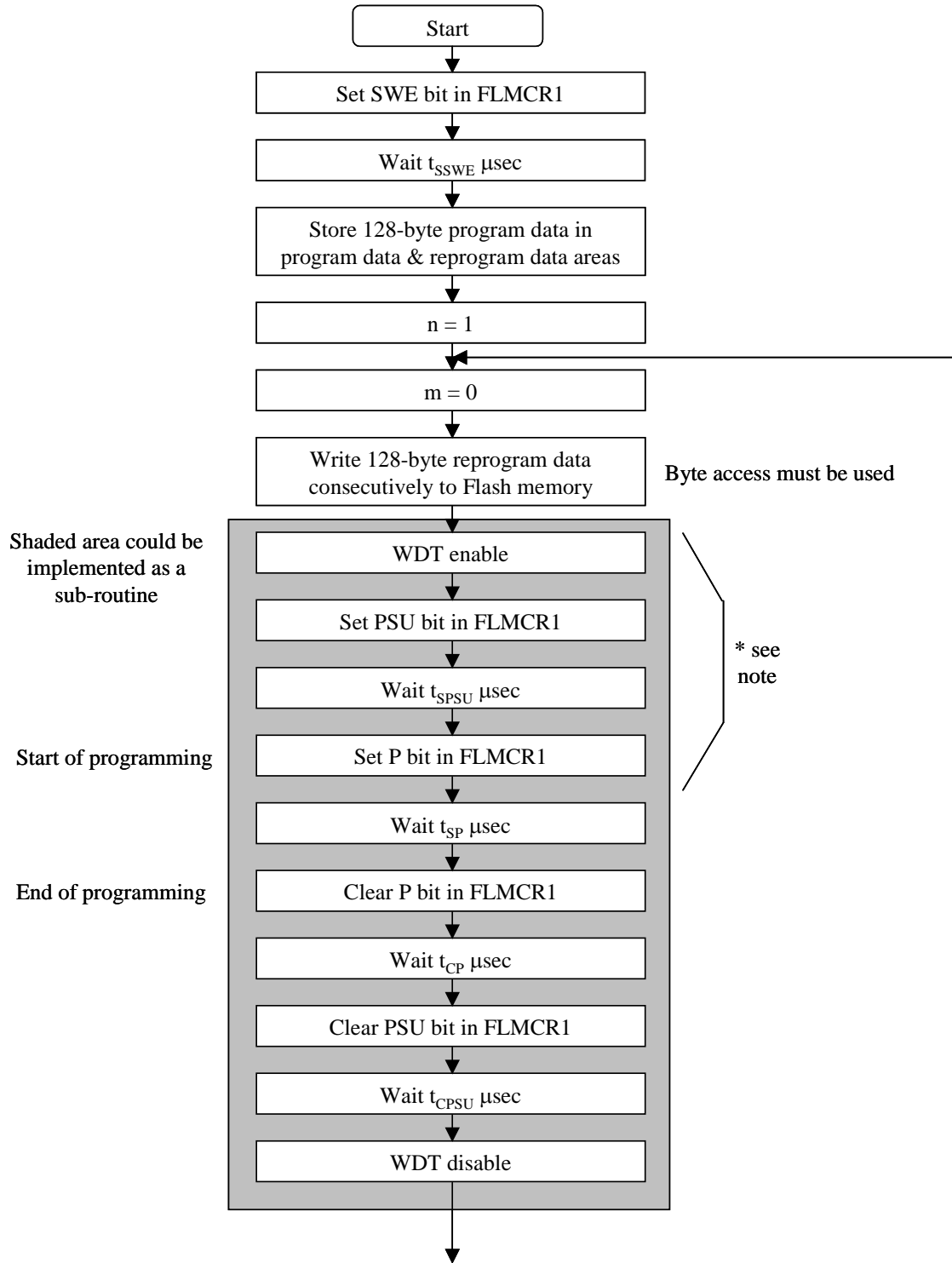
The 0.35 μ m Renesas microcontroller Flash memory has the following characteristics.

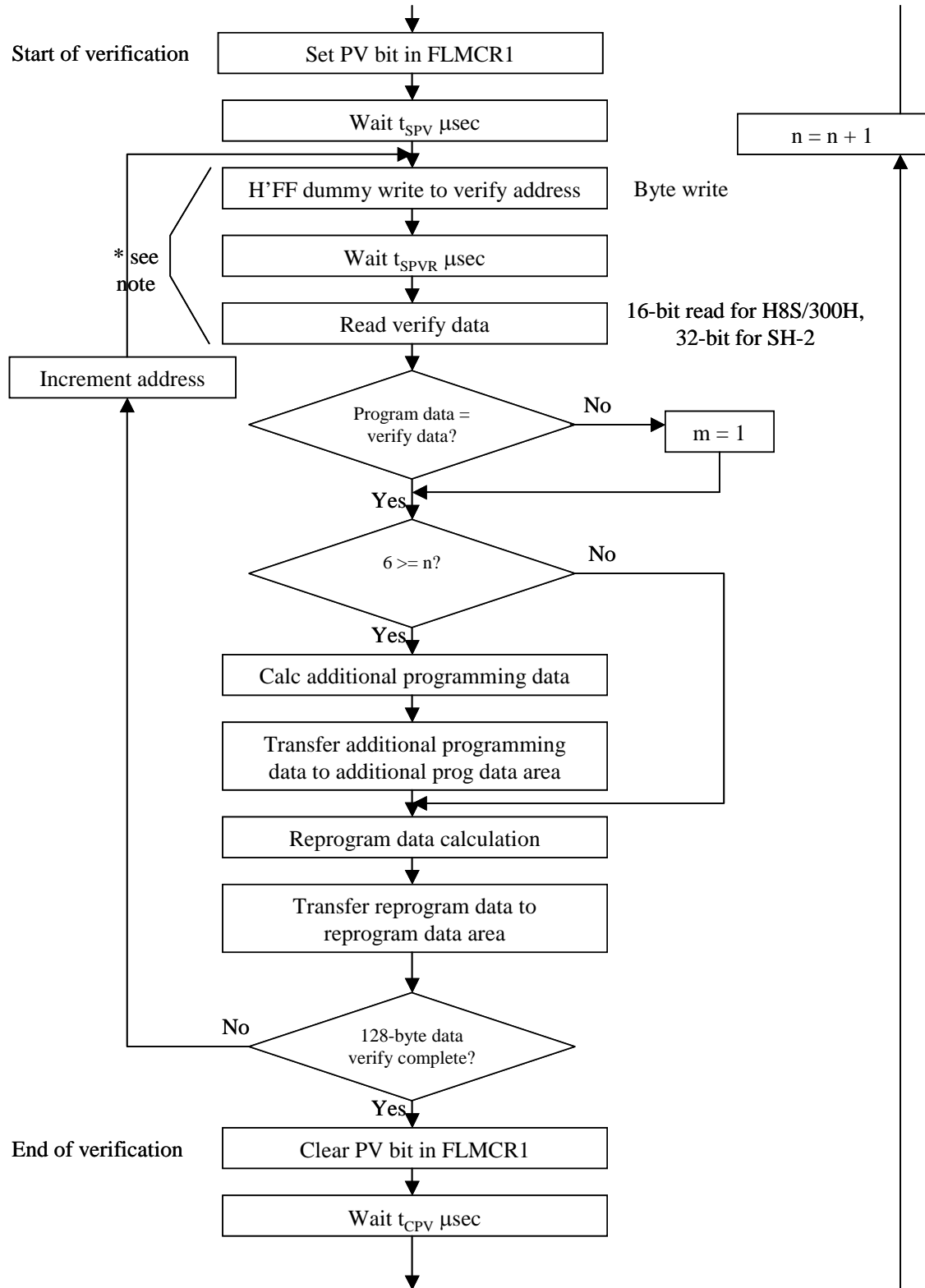
- The Flash memory must be programmed in units of 128 bytes starting on a 128 byte boundary.
- The Flash memory is split into sectors of varying sizes.
- Erasing is performed on a sector by sector basis.
- The erased state is all 1's.
- Programming must be performed in the erased state.
- Programming data is written in 16-bit units for H8(S)(300H) and 32-bits for SH-2.
- Programming and erase verification data is read in 16-bit units for H8S & H8/300H and 32-bits for SH-2.
- Programming times are reduced when compared to the 0.6 μ m Flash memory based Renesas microcontrollers.

Although all 0.35 μ m Renesas Flash microcontrollers essentially have common programming and erasing algorithms it is important that this apps note is read in conjunction with the hardware manual for the device being programmed as there are can subtle differences introduced.

0.35 μ m Program/Program-Verify

Figure 3 shows the typical program/program-verify algorithm for 0.35 μ m Renesas Flash microcontrollers.





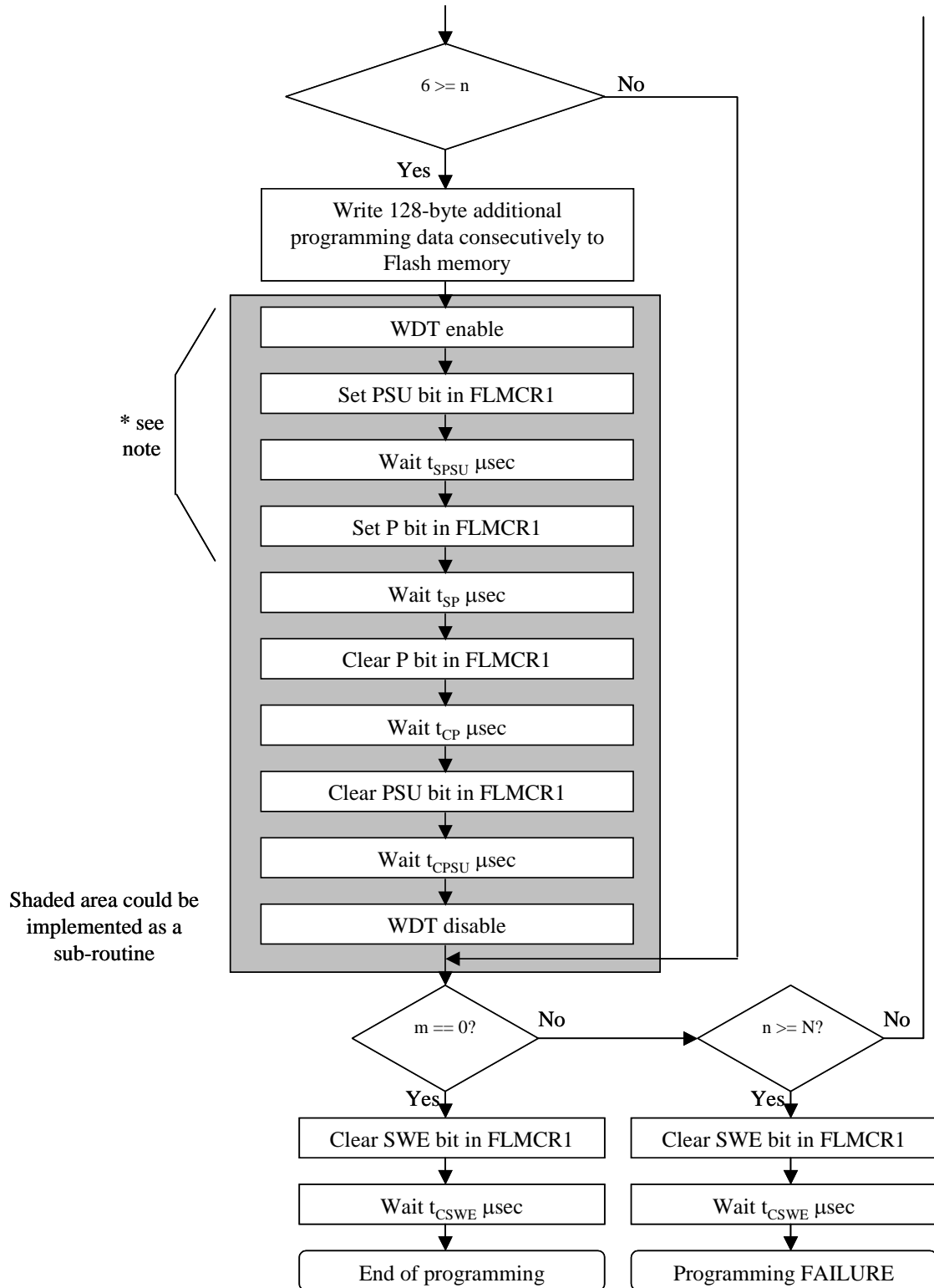


Figure 3: 0.35µm Program/Program-Verify Algorithm

Important aspects of the 0.35µm program/program-verify algorithm worthy of note include:

- The actual values for the delays given in the flowchart should be obtained from the hardware manual for the device being programmed.
- All delay times are minimum times required to allow the internal signals to settle with the exception of the time the 'P' bit is set in the FLMCR1 register. This time is a MAXIMUM value and should not be exceeded.
- Loop counts are maximum values and as such should not be exceeded.
- The verify dummy write should be a byte write of H'FF.
- The verify data read back during the program verify stage must be compared with the actual data to be programmed into the Flash and not the reprogram data or additional program data.
- Programming should only be performed on Flash cells which are in the erased state, '1'.

As can be seen from the algorithm the 0.35µm program/program-verify algorithm is more complex than its 0.6µm counterpart. The program/program-verify process is again a two stage affair with the Flash line being programmed and then verified using the 'weak' read as previously discussed in the 0.6µm section of this apps note. During the programming phase the length of time the 'P' bit in the FLAMCR1 register is set varies depending on how many attempts to program the Flash line have been made. Typically, for the first 6 programming attempts the 'P' bit is set for 30µs and then for the remaining attempts this extends to 200µs. Also, for the first 6 programming attempts after the initial 30µs programming pulse using the reprogramming data there is an extra programming pulse, typically 10µs long, using the additional programming data.

The reprogram data is calculated in the same way as for the 0.6µm algorithm and for completeness is given in table 2 below.

Required Data	Verify Data Read	Reprogram Data
0	0	1
0	1	0
1	0	1
1	1	1

Table 2: Reprogramming Data

The additional programming data used during the first 6 programming attempts is calculated according to the truth table shown in table 3 below.

Reprogram Data	Verify Data Read	Additional Programming Data
0	0	0
0	1	1
1	0	1
1	1	1

Table 3: Additional Programming Data

Appendix C contains C source code for implementing the program/program-verify algorithm described by figure 3 for the H8S series. This code has been tested on an H8S/2612F microcontroller. As with the 0.6µm code this C source should be viewed as example code and modified where necessary to meet the Flash memory programming requirements of a particular Renesas microcontroller. Note should be made that the correct operation of this code is affected by the frequency of the xtal connected to the micro. In this code the xtal frequency is specified as 18.432MHz via the definition 'XTAL' which should be changed to reflect the frequency of the target device. Again the timing delays have been achieved using a hardware timer and so in the case of the shorter delays they can be longer than required but this is not a problem for settling times which have specified minimum values.

Appendix D contains C source code for the SH-2 0.35µm program/program-verify algorithm. This code has been tested on an SH7047F microcontroller.

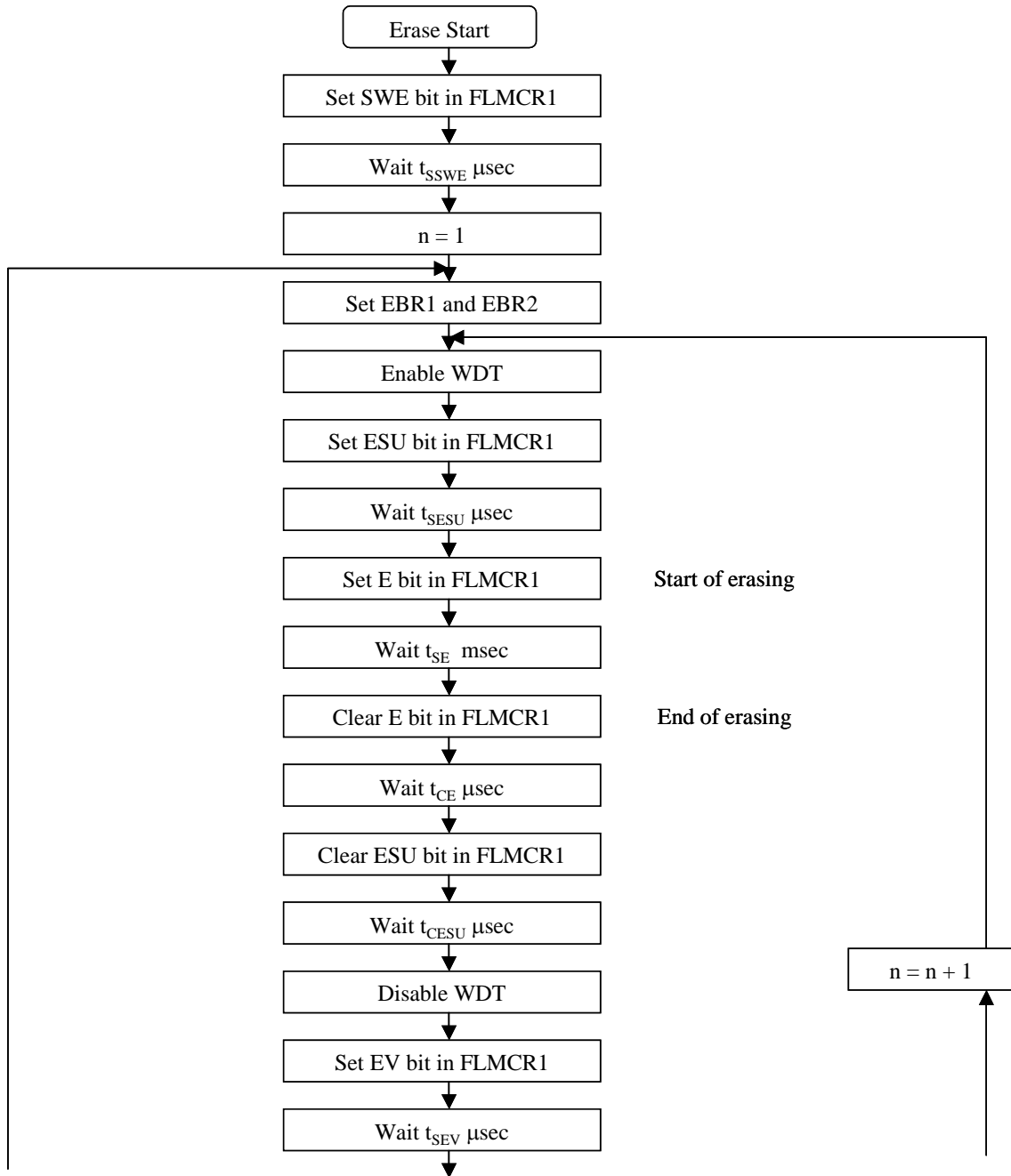
In both instances a 128-byte Flash line can be programmed by calling the function 'prog_flash_line_128' which has the following definition.

```
unsigned char prog_flash_line_128 (unsigned long t_address, union char_rd_datum_union *p_data)
```

The first parameter passed to this function is the start address of the Flash memory to be programmed which must be on a 128-byte boundary. The second passed parameter is a pointer to a 'char_rd_datum_union' union containing the data to be programmed. The function is identical for both H8S and SH-2 with the functionality changing depending on the type specified by the typedef 'read_datum'.

0.35μm Erase/Erase-Verify

Figure 4 below shows the typical erase/erase-verify algorithm for 0.35μm Renesas Flash microcontrollers.



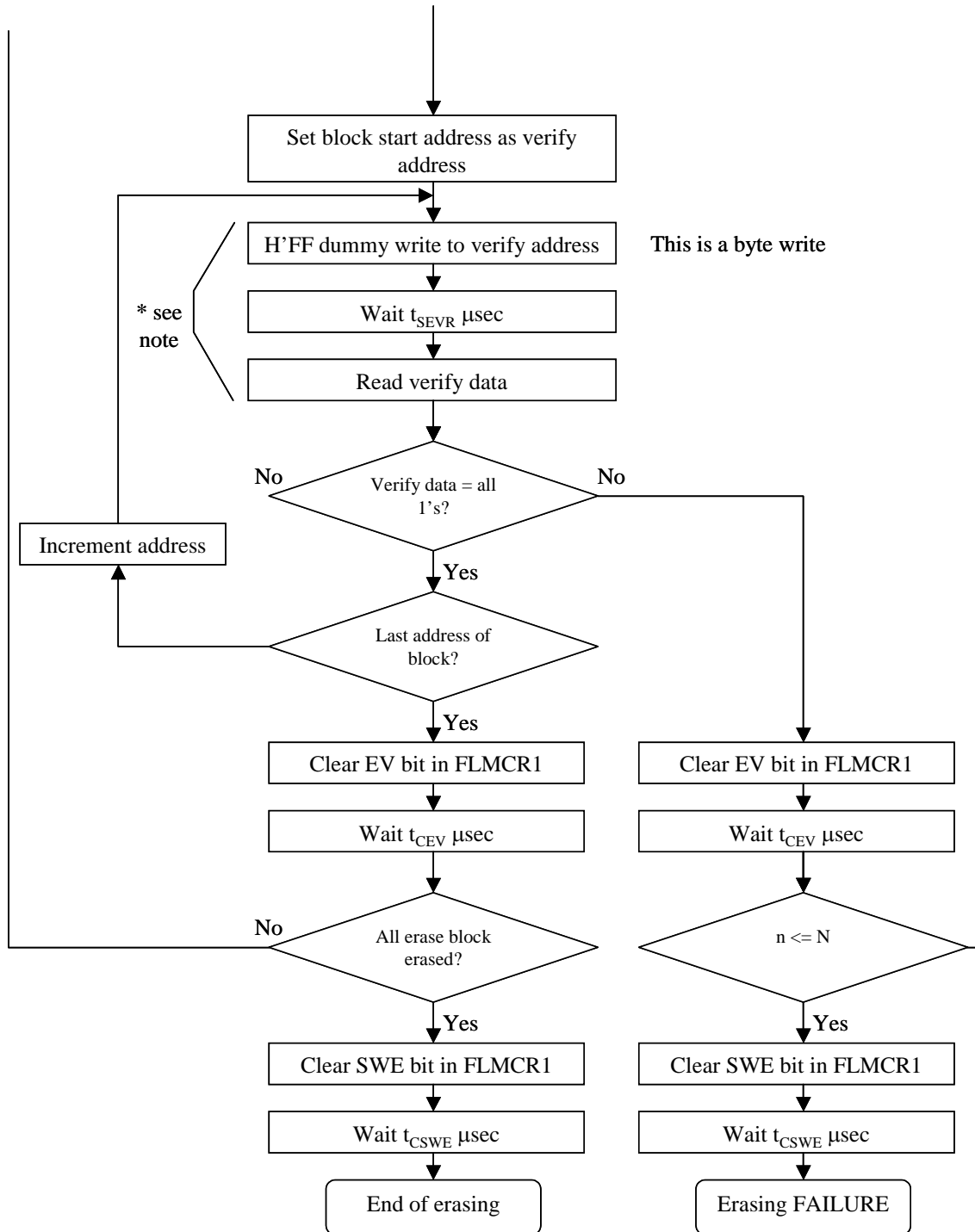


Figure 4: 0.35 μm Erase/Erase-Verify Algorithm

Important aspects of the 0.35µm erase/erase-verify algorithm worthy of note include:

- All delay times are minimum times required to allow the internal signals to settle with one major exception – the time the erase signal (E bit in FLMCR) is set. This time is a MAXIMUM and should not be exceeded.
- The erase pulse time is in units of msec and the settling times in units of µsec.
- Loop counts are maximum values and should not be exceeded.
- When performing a dummy write during the verify stage the dummy write should be performed as a byte access.
- During the verify stage the Flash should be accessed as 16-bits for H8(S)(300H) and 32-bits for SH-2.
- Pre-programming the Flash contents to ‘0’ is not necessary.
- Only one bit in the EBR registers should be set at any one time as each Flash block must be erased separately.

As with 0.6µm Flash erasure the 0.35µm memory is erased in a two stage process. First an attempt is made to erase the Flash block and then the memory is placed into erase-verify mode and its contents read back with a ‘weak’ read and compared with the erase value of all 1s. 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 the Flash memory block is successfully erased or the maximum number of erase attempts specified for the device is reached.

Appendices C and D contain source code listings with functions to erase a specified 0.35µm Flash block for both the H8S/2612F and SH7047F Renesas microcontrollers. The prototype for the erase function is shown below.

```
unsigned char erase_block_035_um (unsigned char block_num);
```

The function should be passed the number of the Flash block to be erased with the first block being numbered ‘0’. A success or failure status byte is returned to the caller. The same function can be used with both H8(S)(300H) and SH-2 based 0.35µm Flash memory so long as the typedef ‘read_datum’ is declared accordingly.

* Important Note Relating to 0.35µm Devices

The Renesas H8/3664F microcontroller, a member of the H8/300H-Tiny family, has a requirement where an ‘RTS’ instruction is not permitted at certain points in the program/program-verify and erase/erase-verify processes. Figures 3 and 4 indicate the points in the algorithm where this is applicable. This impacts the source code provided in appendices C and D as the affected parts of the algorithm feature delays and the code uses a function call to a ‘delay’ function to implement the delay. As the function call eventually results in an ‘RTS’ this will cause problems. A workaround for this problem is to manually inline the ‘delay’ function code in place of the function call at the points highlighted in figures 3 and 4.

Although this is a requirement of the H8/3664F it may not be limited to this device. Therefore, it is strongly recommended that the latest hardware manual is obtained for the microcontroller being used and the Flash algorithms are examined carefully. Failure to do so could permanently damage the microcontroller.

Appendix E contains C source code with modified program/program-verify and erase/erase-verify routines specifically for the H8/3664F. In these routines the 'delay' function calls have been replaced by inline code at the critical points mentioned above. In order to reduce the code size for the H8/3664F implementation a separate 'apply_write_pulse' function has been used. This enables the programming and erasing functionality to comfortably fit in the internal RAM of this device.

0.18 μ m Algorithms

The 0.18 μ m Renesas microcontroller Flash memory has the following characteristics.

- The Flash memory is programmed in units of 128 bytes starting on a 128 byte boundary.
- The erasing and programming routines are built into the device and called from a user application.
- The Flash memory is split into sectors of varying sizes.
- Erasing is performed on a sector by sector basis.
- The erased state is all 1's.
- Programming must only be performed in the erased state.
- Programming times are reduced compared to 0.6 μ m and 0.35 μ m based Renesas microcontrollers.

Although all 0.18 μ m Renesas Flash microcontrollers essentially have common programming and erasing algorithms, it is important that this apps note is read in conjunction with the hardware manual for the device being programmed, as there can be subtle differences introduced.

0.18 μ m Programming

Figure 5 shows the typical programming algorithm for 0.18 μ m Renesas Flash microcontrollers.

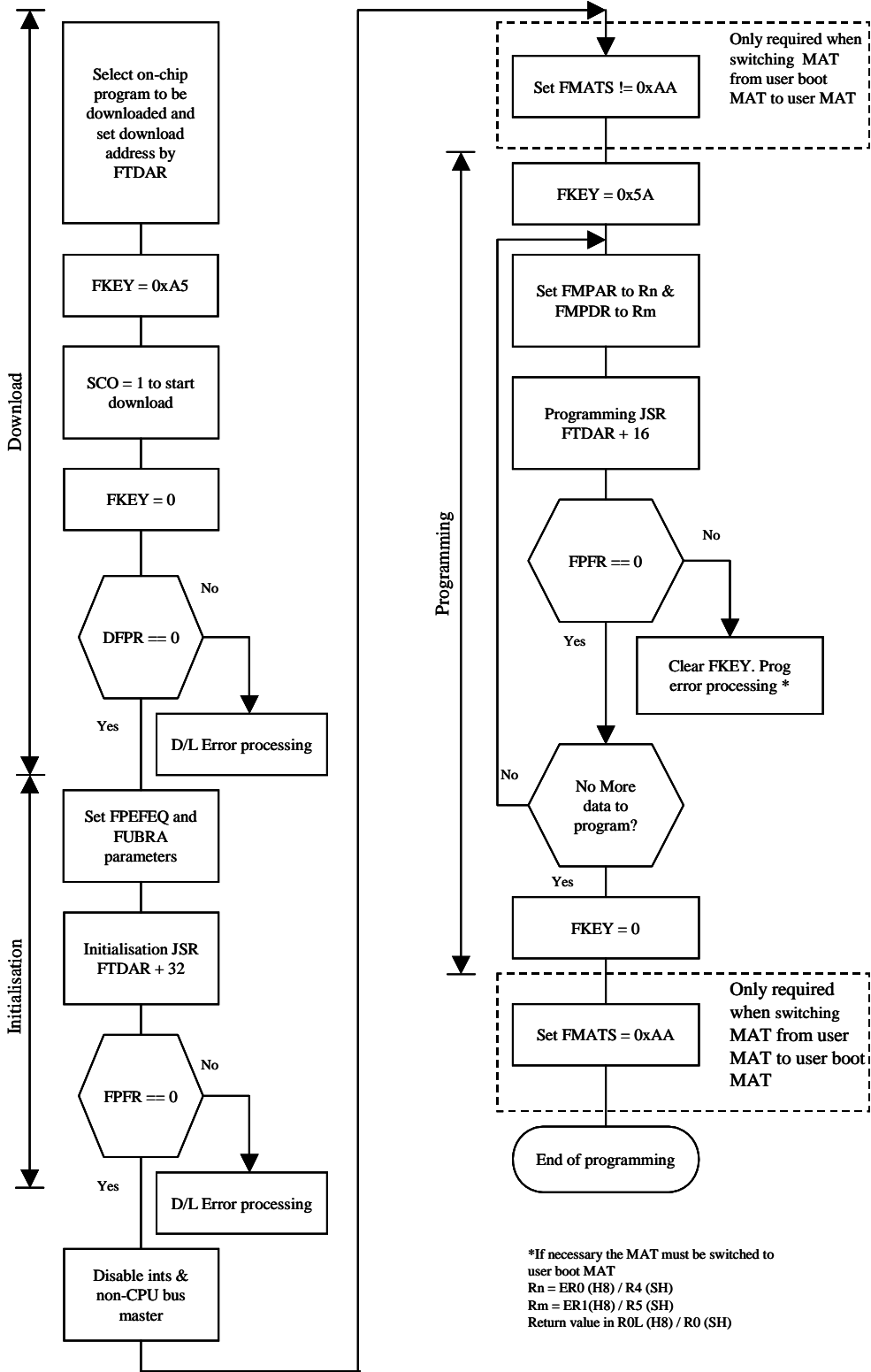


Figure 5: 0.18µm Programming Algorithm

As previously mentioned the actual 0.18µm programming routine is built into the device and is called from a user application. Using the built in programming routine consists of 3 steps – loading, initialisation and programming (execution).

Loading

The loading process copies the built in programming routine into internal RAM for execution. The space used by the programming code is 2000 bytes for H8/300H and 2048 bytes for SH-2. The RAM used by the routine is configurable and set via the FTDAR register. Figure 6 shows the RAM map during the programming process.

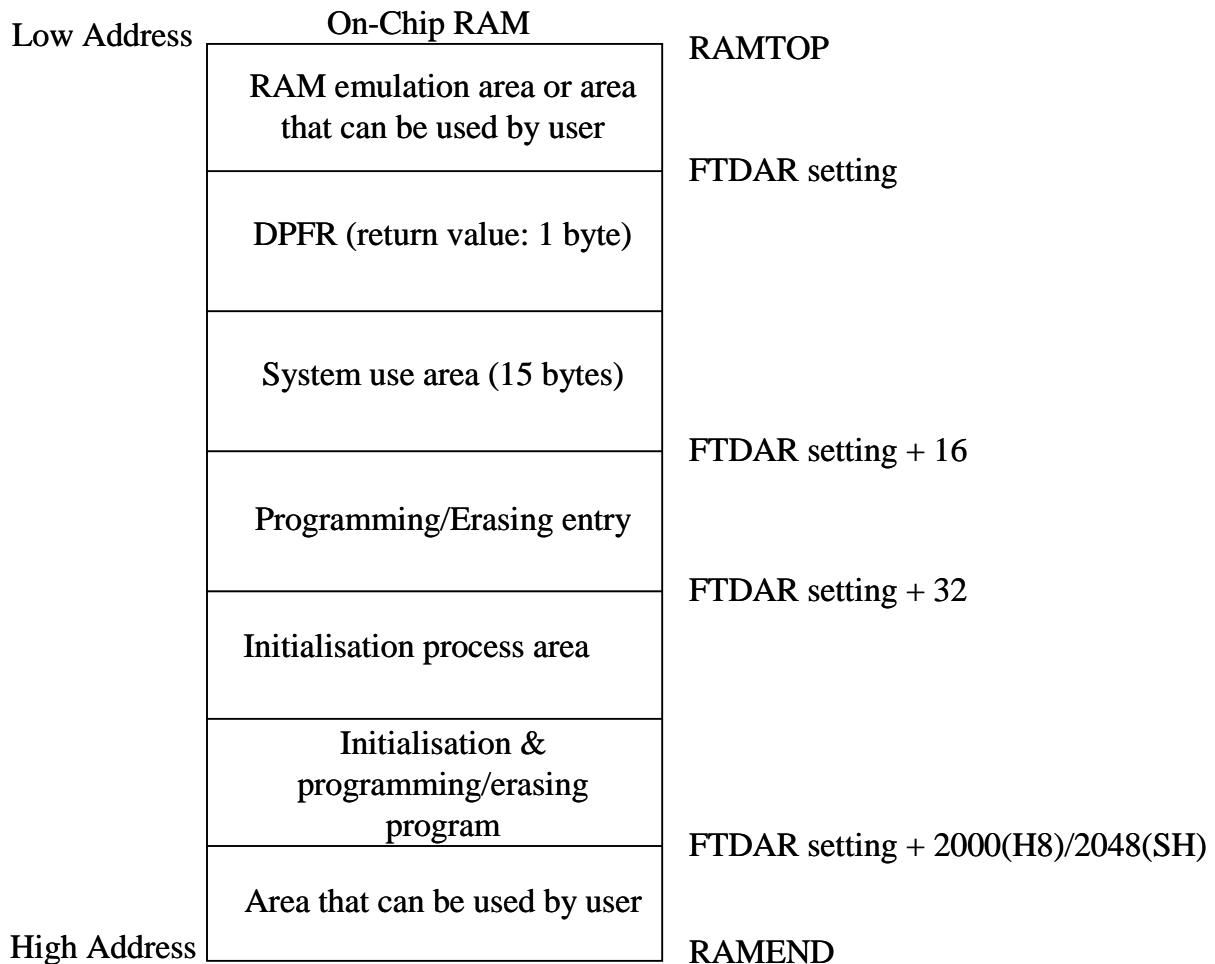


Figure 6: RAM Map During Programming/Erasing

The first byte of this RAM space is given the label DPF_R (download pass/fail result) and is used to indicate the result of the request to download the programming routine to the RAM. The download is executed by setting the routine to be downloaded in the FPCS (flash program code select) and FECS (flash erase code select) registers and then setting SCO (source program copy operation) bit in the FCCS (flash code control and status) register. Four NOPs should be executed after the setting of the SCO bit. When using the Renesas compiler the NOP instruction is inserted in the 'C' code as inline assembly code. With the Renesas compiler, the file containing the inline assembly code must have its output format set as 'assembly code' rather than the default 'machine code'. The DPF_R byte should be initialised to H'FF prior to starting the download process.

The 0.18µm Flash memory offers software protection to prevent accidental programming etc. This protection is implemented using the FKEY register. When this register is set to '0' the protection is active. For downloading the FKEY value should be H'A5 and for programming it should be H'5A. FKEY should be left as zero for the initialisation operation.

The results of the loading request is given in the DPF_R byte. The loading can fail due to incorrect FKEY value, trying to download the program and erase routines at the same time (multi-session) or an invalid setting in the FPCS and FECS registers.

Initialisation

Once the correct routine has successfully been loaded into the internal RAM it must be initialised. The initialisation process configures the routines with the current CPU frequency and user branch address. The user branch option, which is supported by SH-2, allows user code to be called during programming and erasing. This is particularly useful for tickling a watchdog timer during erasing and programming. To use the user branch option the address of the routine should be loaded into the FUBRA (flash user branch address) register. The process of erasing a block or programming a flash line consists of many erase or programming pulses respectively; the user branch routine is called for each such pulse. As the erase and programming pulse lengths are not constant the time between two successive calls of the user branch routine will vary. The minimum and maximum values for this period are given in the Flash memory section of the relevant hardware manual. When the user branch feature is either not supported by the hardware or is not being used the FUBRA register should be set to zero.

The CPU frequency (FPEFEQ) and user branch address (FUBRA) parameters are passed to the programming routine via CPU registers. The actual registers used depend on the device family. For H8/300H, FPEFEQ should be in ER0 and for SH-2 it should be in R4. For the FUBRA value, the registers are ER1 for H8/300H and R5 for SH-2. The FPEFEQ value is the CPU frequency in MHz to 2 decimal places multiplied by 100. For example:

CPU frequency = 20.00MHz

FPEFEQ = 20.00 x 100 = 2000

The FUBRA value is the 32-bit address of the user branch routine.

Although passing these parameters via CPU registers may seem initially inconvenient when programming in 'C', the registers used are those used by the Renesas C/C++ compiler for function parameter passing. Appendices F and G contain source code, in C, for implementing programming and erasing of the 0.18µm Flash memory of the H8/3069F (H8/300H) and SH-2e (SH7058F) respectively. This code contains the function 'func' with the prototype below.

void func (unsigned long ul1, unsigned long ul2);

Passing the FPEFEQ and FUBRA values to this function will result in the values being loaded into the correct CPU registers. With the values in the CPU registers the internal initialisation routine must be called. The start address for this routine is the address set by FTDAR + 32 bytes. In the example code this initialisation routine is called via a function pointer 'fp'. This function returns a byte (FPFR) in R0L (H8/300H) or R0 (SH-2e) containing the result of the initialisation request. A non-zero value indicates that the initialisation has failed. Failure can occur due to the CPU frequency or the user branch address being invalid.

The registers used for parameter passing have been chosen for compatibility with the Renesas C/C++ compiler toolchain. When using other compilers provision must be made to ensure that the correct values are loaded into the correct registers. The KPIT GNUH8 and KPIT GNUSH compilers can be configured to use the Renesas calling convention. If the IAR compiler is being used with H8 then some assembler code will be required.

Programming

With the initialisation completed correctly the 128 byte Flash line can be programmed. If the code is running in user boot mode then, before and after the programming function call, the current MAT must be switched from the user boot MAT to the user MAT and back again. This is achieved by using the FMATS register. Four NOPs should be inserted after changing the FMATS register value.

When programming, the Flash address where programming should start (FMPAR) should be loaded into ER0 for H8/300H and for SH-2 it should be in R4. The address of the data to be programmed (FMPDR), usually in RAM, should be loaded into registers ER1 for H8/300H and R5 for SH-2. The internal programming routine is positioned at address FTDAR + 16. In the example routines programming is executed using the 'fp' function pointer. The return value (FPFR) of this function call contains the result of the programming request. A non-zero value indicates an error such as invalid FWE, invalid FKEY value, incorrect data source address or incorrect data destination address.

If more than one 128 byte Flash line is to be programmed it is not necessary for the programming routine to be downloaded and initialised more than once for each line. This is not implemented in the example source code for reasons of clarity but the download and initialisation functionality can easily be extracted into a subroutine.

The H8/300H 0.18µm makes available an additional feature over the SH-2. This feature is the ability to change the address of the NMI vector for situations where using the NMI interrupt cannot be avoided due to system requirements. The FVACR (Flash vector address control register) enables or disables this feature. When enabled the address of the NMI interrupt service routine should be placed in FVADR (Flash vector address register). This feature is not required by the SH-2 as the whole interrupt and exception vector table can be relocated and then accessed via the VBR (vector base register).

Appendices F and G contain source code, in C, for implementing programming the 0.18 μ m Flash memory of the H8/3069F (H8/300H) and SH-2e (SH7058F) respectively. In both instances a 128 byte Flash line can be programmed by calling the function 'Program018FlashLine' which has the following definition.

```
unsigned short Program018FlashLine( unsigned long Address,  
unsigned char *ProgData );
```

The first parameter passed is the start address of the Flash to be programmed which must be on a 128-byte boundary. The second parameter is a pointer to the data to be programmed into the Flash line. The return value is zero if the Flash line programming was completed successfully. A non-zero value indicates a failure. The error code format is described in the comments at the start of the function.

The source code is supplied in three files for each processor family – 'erase018.c', 'program018.c' and 'flash.h'. The C source files are the same for both H8/3069F and SH7058F. The header file though is different as it contains the specific addresses of the Flash registers and values specific to each device. If the code is to be executed in user boot mode then the definition 'INUSERBOOTMODE' must be defined in order for the MAT switching to be performed. The header files contain extensive comments so there should be no problem in modifying them for use with other 0.18 μ m based Renesas Flash microcontrollers.

0.18µm Erasing

Figure 7 shows the typical erasing algorithm for 0.18µm Renesas Flash microcontrollers.

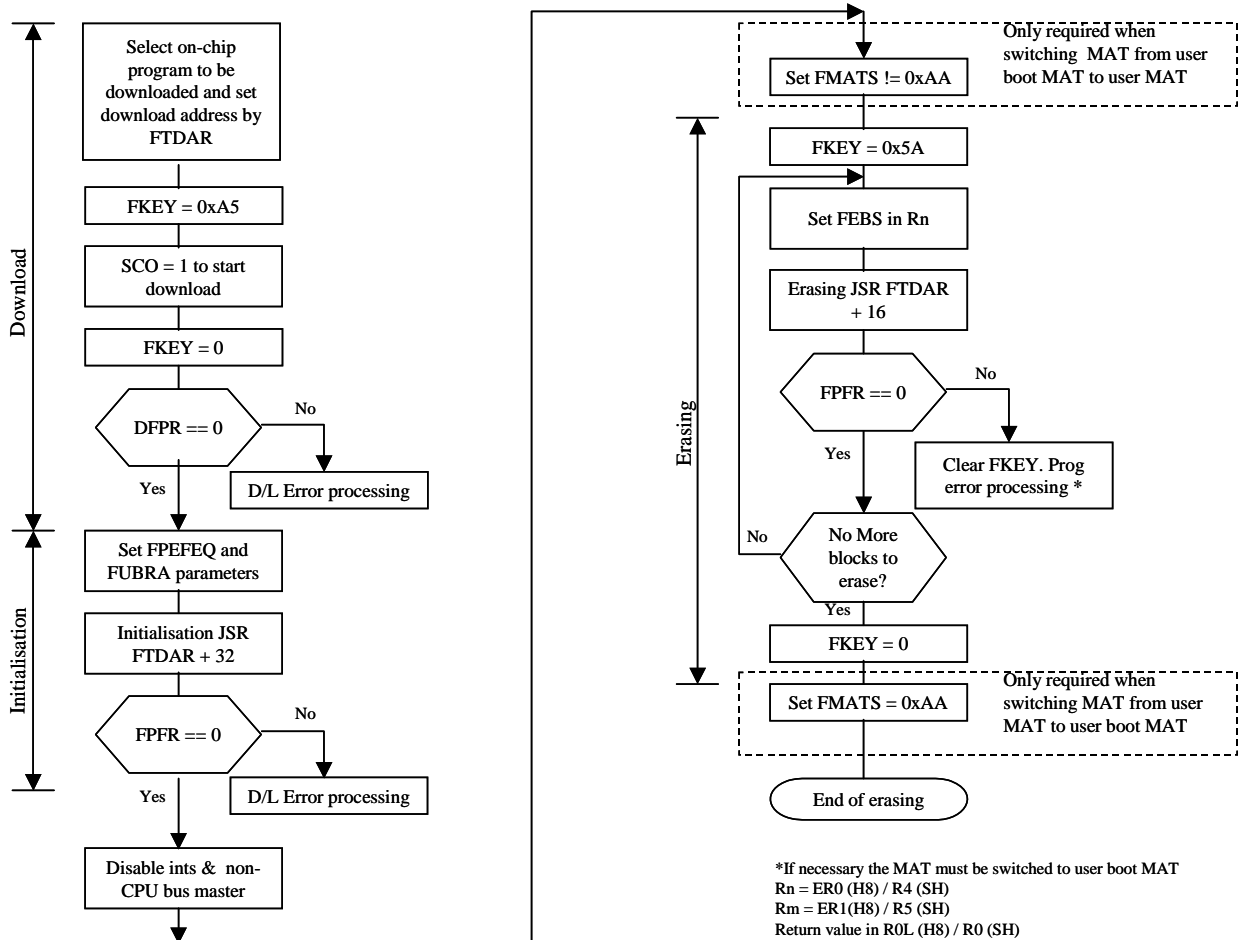


Figure 7: 0.18µm Erasing Algorithm

As previously mentioned the actual 0.18µm erasing routine is built into the device and is called from a user application. Using the built in erasing routine consists of 3 steps – loading, initialisation and erasing (execution).

Loading

The loading of the built in erasing routine into the internal RAM is the same as for the programming routine. The only change is that the erase program is selected in the FPSC and FECS registers. Figure 6 shows the RAM map during the erasing process.

Initialisation

Once the correct routine has successfully been loaded into the internal RAM it must be initialised. The initialisation process for erasing is the same as for the programming routine previously described.

Erasing

With the initialisation completed correctly a Flash block can be erased. If the code is running in user boot mode then, before and after the erasing function call, the current MAT must be switched from the user boot MAT to the user MAT and back again. This is achieved by using the FMATS register. Four NOPs should be inserted after changing the FMATS register value.

The number of the Flash block to be erased (FEBS) should be loaded into ER0 for H8/300H and for SH-2 it should be in R4 using the 'func' function. The internal erasing routine is located at address FTDAR + 16. In the example routines erasing is executed using the 'fp' function pointer. The return value (FPFR) of this function call contains the result of the erasing request. A non-zero value indicates an error such as invalid FWE, invalid FKEY value or invalid erase block.

If more than one erase block is to be erased it is not necessary for the erasing routine to be downloaded and initialised more than once for each block. This is not implemented in the example source code for reasons of clarity but the download and initialisation functionality can easily be extracted into a subroutine.

Again the H8/300H 0.18µm Flash memory NMI vector redirection feature is available during erasing. See the programming section for more details.

Appendices F and G contain source code, in C, for erasing the 0.18µm Flash memory of the H8/3069F (H8/300H) and SH-2e (SH7058F) respectively. In both instances a Flash block can be erased by calling the function 'Erase018FlashBlock' which has the following definition.

```
unsigned short Erase018FlashBlock( unsigned char FlashBlock )
```

The 'FlashBlock' parameter passed is Flash block to be erased which must be valid for the device. The return value is zero if the Flash block erase was completed successfully. A non-zero value indicates a failure. The error code format is described in the comments at the start of the function.

The source code is supplied in three files for each processor family – 'erase018.c', 'program018.c' and 'flash.h'. The C source files are the same for both H8/3069F and SH7058F. The header file though is different as it contains the specific addresses of the Flash registers and values specific to each device. If the code is to be executed in user boot mode then the definition 'INUSERBOOTMODE' must be defined in order for the MAT switching to be performed. The header files contains extensive comments so there should be no problem in modifying them for use with other 0.18µm based Renesas Flash microcontrollers.

Summary

All Renesas microcontrollers with Flash memory have the ability to easily self program and erase their memory.

It is hoped this application note has helped to demystify the process of programming and erasing the Flash memory of Renesas H8 and SH 0.6 μ m, 0.35 μ m and 0.18 μ m microcontrollers. The supplied code examples should provided a basis for implementing custom user mode programming routines giving greater flexibility to current and future applications. It is accepted that the code is not the most efficient in its current form but it is hoped that it is easy to follow. This leaves the user to optimise the code for speed and/or size once an understanding of its operation is established.

APPENDIX A – RENESAS 0.6 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR H8S/2144F

```
// kernel.c
//
//
// Clock speed = 18.432MHz
// H8S2148 uses SCI1 for boot mode
// Kernel start address - 0xffe080

#include "iodefine.h"                // IO header file

// change following define depending on target
// #define SH
#define H8

#ifdef SH
typedef unsigned long read_datum; // unsigned long for SH
#define BLANK_VALUE 0xFFFFFFFF
#else
typedef unsigned short read_datum; // unsigned short for H8S
#define BLANK_VALUE 0xFFFF
#endif

// to get round the problem of different 'iodefine.h' files using slightly
// different names for the flash registers and bits the following defines
// are used
#define FLASH_SWE        FLASH.FLMCR1.BIT.SWE
#define FLASH_PSU2      FLASH.FLMCR2.BIT.PSU
#define FLASH_PSU1      FLASH.FLMCR2.BIT.PSU
#define FLASH_P2        FLASH.FLMCR1.BIT.P
#define FLASH_P1        FLASH.FLMCR1.BIT.P
#define FLASH_PV2      FLASH.FLMCR1.BIT.PV
#define FLASH_PV1      FLASH.FLMCR1.BIT.PV
#define FLASH_EBR1      FLASH.EBR1.BYTE
#define FLASH_EBR2      FLASH.EBR2.BYTE
#define FLASH_EB0       FLASH.EBR2.BIT.EB0
#define FLASH_EB1       FLASH.EBR2.BIT.EB1
#define FLASH_EB2       FLASH.EBR2.BIT.EB2
#define FLASH_EB3       FLASH.EBR2.BIT.EB3
#define FLASH_EB4       FLASH.EBR2.BIT.EB4
#define FLASH_EB5       FLASH.EBR2.BIT.EB5
#define FLASH_EB6       FLASH.EBR2.BIT.EB6
#define FLASH_EB7       FLASH.EBR2.BIT.EB7
#define FLASH_EB8       FLASH.EBR1.BIT.EB8
#define FLASH_EB9       FLASH.EBR1.BIT.EB9
#define FLASH_EB10      FLASH.EBR1.BIT.EB9
#define FLASH_EB11      FLASH.EBR1.BIT.EB9
#define FLASH_ESU2      FLASH.FLMCR2.BIT.ESU
#define FLASH_ESU1      FLASH.FLMCR2.BIT.ESU
#define FLASH_E2        FLASH.FLMCR1.BIT.E
#define FLASH_E1        FLASH.FLMCR1.BIT.E
#define FLASH_EV2       FLASH.FLMCR1.BIT.EV
#define FLASH_EV1       FLASH.FLMCR1.BIT.EV
```



```

// H8S2148 specific
#define MAX_FLASH_ADDR      0x20000
#define FLASH_LINE_SIZE    32
#define NO_OF_FLASH_BLOCKS  10
#define XTAL                 18432000L
#define MAX_PROG_COUNT      1000
#define MAX_ERASE_ATTEMPTS  120
#define MAX_FLMCR1_ADDRESS  0x1FFFFL

// 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)
const unsigned long eb_block_addr [NO_OF_FLASH_BLOCKS + 1] = {
0x00000000L,
0x00000400L,
0x00000800L,
0x00000C00L,
0x00001000L,
0x00008000L,
0x0000C000L,
0x0000E000L,
0x00010000L,
0x00018000L,
0x00020000L /* 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
#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 FIFTY_USEC            ((5L * XTAL) / 800000L)
#define TWO_HUNDRED_USEC      ((2L * XTAL) / 80000L)
#define FIVE_MSEC             ((5L * XTAL) / 8000L)

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

// function prototypes
unsigned char prog_flash_line_32 (unsigned long t_address, union char_rd_datum_union *p_data);
void delay (unsigned short);
void init_delay_timer (void);
unsigned char erase_block_06_um (unsigned char block_num);

// variables
volatile unsigned long delay_counter;

```

```

// Functions
unsigned char prog_flash_line_32 (unsigned long t_address, union char_rd_datum_union *p_data)
{
    unsigned short n_prog_count;    // loop counter for programming attempts (0->MAX_PROG_COUNT)
    unsigned short d;              // general variable used for various loop counts
    unsigned char m;               // flag to indicate if re-programming required 1=yes 0=no
    unsigned char *dest_address;   // pointer used for writing to the flash
    unsigned char *uc_v_write_address; // pointer used for writing to the addr to be verified
    read_datum *ul_v_read_address; // pointer used to read address being verified
    unsigned char ax;              // variable used as loop counter for incrementing the
                                   // pointer to the byte being written next

in verify process
    union char_rd_datum_union reprog_data; // storage (on stack) for the re-program data

    // enable flash writes
    FLASH_SWE = 1;

    // wait 10us
    delay (TEN_USEC);

    // copy data from program data area to reprogram data area
    for (d=0; d<FLASH_LINE_SIZE; d++)
    {
        reprog_data.c[d] = p_data->c[d];
    }

    // program the data in FLASH_LINE_SIZE 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
        dest_address = (unsigned char *) t_address;
        for (d=0; d<FLASH_LINE_SIZE; d++)
        {
            *dest_address++ = reprog_data.c[d];
        }

        // enter program setup
        if ( t_address > MAX_FLMCR1_ADDRESS )
        {
            // FLMCR2
            FLASH_PSU2 = 1;
        }
        else
        {
            // FLMCR1
            FLASH_PSU1 = 1;
        }

        // wait 50us
        delay (FIFTY_USEC);

        // start programming pulse
        if ( t_address > MAX_FLMCR1_ADDRESS )
        {
            // FLMCR2

```

```

        FLASH_P2 = 1;
    }
    else
    {
        // FLMCR1
        FLASH_P1 = 1;
    }

    // wait 200us
    delay (TWO_HUNDRED_USEC);

    // stop programming pulse
    if ( t_address > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_P2 = 0;
    }
    else
    {
        // FLMCR1
        FLASH_P1 = 0;
    }

    // wait 20us
    delay (TEN_USEC);

    // leave programming setup
    if ( t_address > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_PSU2 = 0;
    }
    else
    {
        // FLMCR1
        FLASH_PSU1 = 0;
    }

    // wait 10us
    delay (TEN_USEC);

    // enter program verify mode
    if ( t_address > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_PV2 = 1;
    }
    else
    {
        // FLMCR1
        FLASH_PV1 = 1;
    }

    // wait 4us
    delay (FOUR_USEC);

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

```

```

ul_v_read_address = (read_datum *) t_address;

// 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;

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

    // wait 2us
    delay (TWO_USEC);

    // 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;
    }

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

    // increment the pointers
    ul_v_read_address++;
} // end of verify loop

// exit program verify mode
if ( t_address > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_PV2 = 0;
}
else
{
    // FLMCR1
    FLASH_PV1 = 0;
}

// wait 4us
delay (FOUR_USEC);

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

    // end of successful programming

```

```

        return (PROG_PASS);
    }

} // end of MAX_PROG_COUNT attempts to program

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

// end of failed programming
return (PROG_FAIL);
}

unsigned char erase_block_06_um (unsigned char block_num)
{
    unsigned char erase;          // flag showing erase status - either BLANK or NOT_BLANK
    unsigned long attempts;      // counter for erase attempts (0->MAX_ERASE_ATTEMPTS)
    read_datum *ul_v_read;      // pointer for reading erase/verify data
    unsigned char *uc_v_write;   // pointer for writing erase/verify dummy byte
    unsigned char inc_uc_v_write_count; // loop counter for incrementing the uc_v_write variable

    // 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 flash writes
        FLASH_SWE = 1;

        // wait 10us
        delay (TEN_USEC);

        // set the correct EB bit in correct EBR register
        FLASH_EBR1 = 0;
        FLASH_EBR2 = 0;
        switch (block_num)
        {
            case 0:
                FLASH_EB0 = 1;
                break;

            case 1:
                FLASH_EB1 = 1;
                break;

            case 2:
                FLASH_EB2 = 1;
                break;
        }
    }
}

```

```

    case 3:
        FLASH_EB3 = 1;
    break;

    case 4:
        FLASH_EB4 = 1;           // note the change to EBR2 here!
    break;

    case 5:
        FLASH_EB5 = 1;
    break;

    case 6:
        FLASH_EB6 = 1;
    break;

    case 7:
        FLASH_EB7 = 1;
    break;

    case 8:
        FLASH_EB8 = 1;
    break;

    case 9:
        FLASH_EB9 = 1;
    break;

    case 10:
        FLASH_EB10 = 1;
    break;

    case 11:
        FLASH_EB11 = 1;
    break;
}

// initialise the attempts counter
// 0 as we check for less than MAX (not <= MAX)
attempts = 0;
erase = NOT_BLANK;
while ( ( attempts < MAX_ERASE_ATTEMPTS ) && ( erase == NOT_BLANK ) )
{
    // increment the attempts counter
    attempts++;

    // enter erase setup mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_ESU2 = 1;
    }
    else
    {
        // FLMCR1
        FLASH_ESU1 = 1;
    }
}

```

```

// wait 200us
delay (TWO_HUNDRED_USEC);

// transition to erase mode
if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_E2 = 1;
}
else
{
    // FLMCR1
    FLASH_E1 = 1;
}

// wait 5ms
delay (FIVE_MSEC);

// exit erase mode
if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_E2 = 0;
}
else
{
    // FLMCR1
    FLASH_E1 = 0;
}

// wait 10us
delay (TEN_USEC);

// exit erase setup mode
if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_ESU2 = 0;
}
else
{
    // FLMCR1
    FLASH_ESU1 = 0;
}

// wait 10 us
delay (TEN_USEC);

// enter erase/verify mode
if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_EV2 = 1;
}
else
{
    // FLMCR1
    FLASH_EV1 = 1;
}

```

```

    }

    // wait 20 us
    delay (TWENTY_USEC);

    // verify flash has been erased
    // read all the addresses in the current erase block and check that they are
    // successfully erased
    // exit this loop if a non-erased address is detected
    ul_v_read = (read_datum *) eb_block_addr [block_num];
    uc_v_write = (unsigned char *) eb_block_addr [block_num];
    erase = BLANK;
    while ( (erase == BLANK) && ( ul_v_read < (read_datum *) eb_block_addr
[block_num + 1] ) )
    {
        // dummy write
        *uc_v_write = 0xff;

        // wait 2 us
        delay (TWO_USEC);

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

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

    // exit erase/verify mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_EV2 = 0;
    }
    else
    {
        // FLMCR1
        FLASH_EV1 = 0;
    }

    // wait 5 us
    delay (FIVE_USEC);
} // end of outer while loop

// end either of erase attempts or block has been erased ok
//

```



```

        // disable flash writes
        FLASH_SWE = 0;

        // check if block has been erased ok
        if (erase == BLANK)
        {
            // successfully erased
            return ERASE_PASS;
        }
        else
        {
            // failed to erase this block
            return ERASE_FAIL;
        }
    }
}

void init_delay_timer (void)
{
    MSTPCR.BIT.B13 = 0;          // FRT enabled in Module Stop Register

    FRT.TOCR.BIT.OCRS = 0;      // Access to OCRA
    FRT.OCRA = 0;
    FRT.TOCR.BIT.OCRS = 1;      // Access to OCRB
    FRT.OCRB = 0;

    FRT.TOCR.BIT.ICRS = 0;      // Access to ICRA, ICRB & ICRC enabled
    FRT.ICRA = 0;
    FRT.ICRB = 0;
    FRT.ICRC = 0;
    FRT.ICRD = 0;

    FRT.TOCR.BIT.ICRS = 1;      // Access to OCRAR, OCRAF & OCRDM enabled
    FRT.OCRAR = 0;
    FRT.OCRAF = 0;
    FRT.OCRDM = 0;

    FRT.TIER.BYTE = 0;          // Disable all FRT interrupts

    FRT.TCSR.BIT.CCLRA = 1;      // 0 = Timer NOT cleared by compare-match A
                                   // 1 = Timer IS cleared by compare-match A

    FRT.TCR.BIT.IEDGA = 0;      // Capture on falling edge
    FRT.TCR.BIT.IEDGB = 0;      // Capture on falling edge
    FRT.TCR.BIT.IEDGC = 0;      // Capture on falling edge
    FRT.TCR.BIT.IEDGD = 0;      // Capture on falling edge
    FRT.TCR.BIT.BUFEA = 0;      // ICRC not used as buffer for I/C A
    FRT.TCR.BIT.BUFEB = 0;      // ICRC not used as buffer for I/C B
    FRT.TCR.BIT.CKS = 1;        // Clock source: CKS1 = 0 CKS0 = 1 ( clk / 8 )

    FRT.TOCR.BIT.ICRDMS = 0;
    FRT.TOCR.BIT.OCRAMS = 0;
    FRT.TOCR.BIT.OEA = 0;
    FRT.TOCR.BIT.OEB = 0;
    FRT.TOCR.BIT.OLVLA = 0;
    FRT.TOCR.BIT.OLVLB = 0;
}

```

```
void delay (unsigned short d)
{
    FRT.TOCR.BIT.OCRS = 0;           // Access to OCRA
    FRT.OCRA = d;                   // set compare value
    FRT.FRC = 0;                    // clear TCNT to 0
    FRT.TCSR.BIT.OCFA = 0;         // Clear flag
    while(FRT.TCSR.BIT.OCFA == 0);  // wait until compare value is met
}

void main (void)
{
}
```

APPENDIX B – RENESAS 0.6 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR SH7045F

```
// kernel.c
//
// Programming kernel for SH7045F
//
// Clock speed = 29.488MHz

#include "iodefine.h"                // IO header file

// change following define depending on target
#define SH
// #define H8

#ifdef SH
typedef unsigned long read_datum; // unsigned long for SH
#define BLANK_VALUE                0xFFFFFFFF
#else
typedef unsigned short read_datum; // unsigned short for H8S
#define BLANK_VALUE                0xFFFF
#endif

// to get round the problem of different 'iodefine.h' files using slightly
// different names for the flash registers and bits the following defines
// are used
#define FLASH_SWE                   FLASH.FLMCR1.BIT.SWE
#define FLASH_PSU2                  FLASH.FLMCR2.BIT.PSU2
#define FLASH_PSU1                  FLASH.FLMCR1.BIT.PSU1
#define FLASH_P2                    FLASH.FLMCR2.BIT.P2
#define FLASH_P1                    FLASH.FLMCR1.BIT.P1
#define FLASH_PV2                   FLASH.FLMCR2.BIT.PV2
#define FLASH_PV1                   FLASH.FLMCR1.BIT.PV1
#define FLASH_EB0                   FLASH.EBR1.BIT.EB0
#define FLASH_EB1                   FLASH.EBR1.BIT.EB1
#define FLASH_EB2                   FLASH.EBR1.BIT.EB2
#define FLASH_EB3                   FLASH.EBR1.BIT.EB3
#define FLASH_EB4                   FLASH.EBR2.BIT.EB4
#define FLASH_EB5                   FLASH.EBR2.BIT.EB5
#define FLASH_EB6                   FLASH.EBR2.BIT.EB6
#define FLASH_EB7                   FLASH.EBR2.BIT.EB7
#define FLASH_EB8                   FLASH.EBR2.BIT.EB8
#define FLASH_EB9                   FLASH.EBR2.BIT.EB9
#define FLASH_EB10                  FLASH.EBR2.BIT.EB10
#define FLASH_EB11                  FLASH.EBR2.BIT.EB11
#define FLASH_ESU2                  FLASH.FLMCR2.BIT.ESU2
#define FLASH_ESU1                  FLASH.FLMCR1.BIT.ESU1
#define FLASH_E2                    FLASH.FLMCR2.BIT.E2
#define FLASH_E1                    FLASH.FLMCR1.BIT.E1
#define FLASH_EV2                   FLASH.FLMCR2.BIT.EV2
#define FLASH_EV1                   FLASH.FLMCR1.BIT.EV1
#define FLASH_EBR1                  FLASH.EBR1.BYTE
#define FLASH_EBR2                  FLASH.EBR2.BYTE

// SH704/5F specific
```

```

#define MAX_FLASH_ADDR                0x40000
#define FLASH_LINE_SIZE                32
#define NO_OF_FLASH_BLOCKS            12
#define XTAL                            29488000L
#define MAX_PROG_COUNT                1000
#define MAX_ERASE_ATTEMPTS            60
#define MAX_FLMCR1_ADDRESS            0x1FFFFL
// 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)
const unsigned long eb_block_addr [NO_OF_FLASH_BLOCKS + 1] = {
    0x00000000L,
    0x00008000L,
    0x00010000L,
    0x00018000L,
    0x00020000L,
    0x00028000L,
    0x00030000L,
    0x00038000L,
    0x0003F000L,
    0x0003F400L,
    0x0003F800L,
    0x0003FC00L,
    0x00040000L          /* 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 SH7045F CMT with a system clock divider of 8
#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 FIFTY_USEC                       ((5L * XTAL) / 800000L)
#define TWO_HUNDRED_USEC                 ((2L * XTAL) / 80000L)
#define FIVE_MSEC                        ((5L * XTAL) / 8000L)

// function prototypes
void main (void);
unsigned char prog_flash_line_32 (unsigned long t_address, union char_rd_datum_union *p_data);
void delay (unsigned short);
void init_delay_timer (void);
unsigned char erase_block_06_um (unsigned char block_num);

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

```

```

// variables
volatile unsigned long delay_counter;

// Functions
unsigned char prog_flash_line_32 (unsigned long t_address, union char_rd_datum_union *p_data)
{
    unsigned short n_prog_count;    // loop counter for programming attempts (0->MAX_PROG_COUNT)
    unsigned short d;                // general variable used for various loop counts
    unsigned char m;                 // flag to indicate if re-programming required 1=yes 0=no
    unsigned char *dest_address;     // pointer used for writing to the flash
    unsigned char *uc_v_write_address; // pointer used for writing to the addr to be verified
    read_datum *ul_v_read_address; // pointer used to read address being verified
    unsigned char ax;                // variable used as loop counter for incrementing the
                                     // pointer to the byte being written next

in verify process
    union char_rd_datum_union reprog_data; // storage (on stack) for the re-program data

    // enable flash writes
    FLASH_SWE = 1;

    // wait 10us
    delay (TEN_USEC);

    // copy data from program data area to reprogram data area
    for (d=0; d<FLASH_LINE_SIZE; d++)
    {
        reprog_data.c[d] = p_data->c[d];
    }

    // program the data in FLASH_LINE_SIZE 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
        dest_address = (unsigned char *) t_address;
        for (d=0; d<FLASH_LINE_SIZE; d++)
        {
            *dest_address++ = reprog_data.c[d];
        }

        // enter program setup
        if ( t_address > MAX_FLMCR1_ADDRESS )
        {
            // FLMCR2
            FLASH_PSU2 = 1;
        }
        else
        {
            // FLMCR1
            FLASH_PSU1 = 1;
        }

        // wait 50us
        delay (FIFTY_USEC);

        // start programming pulse
    }
}

```

```

if ( t_address > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_P2 = 1;
}
else
{
    // FLMCR1
    FLASH_P1 = 1;
}

// wait 200us
delay (TWO_HUNDRED_USEC);

// stop programming pulse
if ( t_address > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_P2 = 0;
}
else
{
    // FLMCR1
    FLASH_P1 = 0;
}

// wait 20us
delay (TEN_USEC);

// leave programming setup
if ( t_address > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_PSU2 = 0;
}
else
{
    // FLMCR1
    FLASH_PSU1 = 0;
}

// wait 10us
delay (TEN_USEC);

// enter program verify mode
if ( t_address > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_PV2 = 1;
}
else
{
    // FLMCR1
    FLASH_PV1 = 1;
}

// wait 4us
delay (FOUR_USEC);

```

```

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

// 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;

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

    // wait 2us
    delay (TWO_USEC);

    // 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;
    }

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

    // increment the pointers
    ul_v_read_address++;
} // end of verify loop

// exit program verify mode
if ( t_address > MAX_FLMCR1_ADDRESS )
{
    // FLMCR2
    FLASH_PV2 = 0;
}
else
{
    // FLMCR1
    FLASH_PV1 = 0;
}

// wait 4us
delay (FOUR_USEC);

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

```

```

        FLASH_SWE = 0;

        // end of successful programming
        return (PROG_PASS);
    }

} // end of MAX_PROG_COUNT attempts to program

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

// end of failed programming
return (PROG_FAIL);
}

unsigned char erase_block_06_um (unsigned char block_num)
{
    unsigned char erase;          // flag showing erase status - either BLANK or NOT_BLANK
    unsigned long attempts;      // counter for erase attempts (0->MAX_ERASE_ATTEMPTS)
    read_datum *ul_v_read;      // pointer for reading erase/verify data
    unsigned char *uc_v_write;   // pointer for writing erase/verify dummy byte
    unsigned char inc_uc_v_write_count; // loop counter for incrementing the uc_v_write variable

    // 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 flash writes
        FLASH_SWE = 1;

        // wait 10us
        delay (TEN_USEC);

        // set the correct EB bit in correct EBR register
        FLASH_EBR1 = 0;
        FLASH_EBR2 = 0;
        switch (block_num)
        {
            case 0:
                FLASH_EB0 = 1;
                break;

            case 1:
                FLASH_EB1 = 1;
                break;

            case 2:

```



```

        FLASH_EB2 = 1;
    break;

    case 3:
        FLASH_EB3 = 1;
    break;

    case 4:
        FLASH_EB4 = 1;           // note the change to EBR2 here!
    break;

    case 5:
        FLASH_EB5 = 1;
    break;

    case 6:
        FLASH_EB6 = 1;
    break;

    case 7:
        FLASH_EB7 = 1;
    break;

    case 8:
        FLASH_EB8 = 1;
    break;

    case 9:
        FLASH_EB9 = 1;
    break;

    case 10:
        FLASH_EB10 = 1;
    break;

    case 11:
        FLASH_EB11 = 1;
    break;
}

// initialise the attempts counter
// 0 as we check for less than MAX (not <= MAX)
attempts = 0;
erase = NOT_BLANK;
while ( ( attempts < MAX_ERASE_ATTEMPTS ) && ( erase == NOT_BLANK ) )
{
    // increment the attempts counter
    attempts++;

    // enter erase setup mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_ESU2 = 1;
    }
    else
    {
        // FLMCR1

```

```

        FLASH_ESU1 = 1;
    }

    // wait 200us
    delay (TWO_HUNDRED_USEC);

    // transition to erase mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_E2 = 1;
    }
    else
    {
        // FLMCR1
        FLASH_E1 = 1;
    }

    // wait 5ms
    delay (FIVE_MSEC);

    // exit erase mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_E2 = 0;
    }
    else
    {
        // FLMCR1
        FLASH_E1 = 0;
    }

    // wait 10us
    delay (TEN_USEC);

    // exit erase setup mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_ESU2 = 0;
    }
    else
    {
        // FLMCR1
        FLASH_ESU1 = 0;
    }

    // wait 10 us
    delay (TEN_USEC);

    // enter erase/verify mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_EV2 = 1;
    }
    else

```

```

    {
        // FLMCR1
        FLASH_EV1 = 1;
    }

    // wait 20 us
    delay (TWENTY_USEC);

    // verify flash has been erased
    // read all the addresses in the current erase block and check that they are
    // successfully erased
    // exit this loop if a non-erased address is detected
    ul_v_read = (read_datum *) eb_block_addr [block_num];
    uc_v_write = (unsigned char *) eb_block_addr [block_num];
    erase = BLANK;
    while ( (erase == BLANK) && ( ul_v_read < (read_datum *) eb_block_addr
[block_num + 1] ) )
    {
        // dummy write
        *uc_v_write = 0xff;

        // wait 2 us
        delay (TWO_USEC);

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

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

    // exit erase/verify mode
    if ( eb_block_addr [block_num] > MAX_FLMCR1_ADDRESS )
    {
        // FLMCR2
        FLASH_EV2 = 0;
    }
    else
    {
        // FLMCR1
        FLASH_EV1 = 0;
    }

    // wait 5 us
    delay (FIVE_USEC);
} // end of outer while loop

```

```

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

        // check if block has been erased ok
        if (erase == BLANK)
        {
            // successfully erased
            return ERASE_PASS;
        }
        else
        {
            // failed to erase this block
            return ERASE_FAIL;
        }
    }
}

void init_delay_timer (void)
{
    // initialises compare match timer (CMT) channel 0

    // enable in module stop register
    //MST.MSTCR2.BIT.MSTP12 = 0;

    // stop channel 0
    CMT.CMSTR.BIT.STR0 = 0;

    // channel 0 compare match interrupt disabled
    CMT0.CMCSR.BIT.CMIE = 0;

    // system clock / 8
    CMT0.CMCSR.BIT.CKS = 0;

    // start timer
    CMT.CMSTR.BIT.STR0 = 1;
}

void delay (unsigned short d)
{
    // load compare match value into the constant register
    CMT0.CMCOR = d;

    // clear counter
    CMT0.CMCNT = 0;

    // clear compare match flag
    CMT0.CMCSR.BIT.CMF = 0;

    // loop until we have a compare match
    while (CMT0.CMCSR.BIT.CMF == 0);
}

void main (void)
{
}

```

APPENDIX C – RENESAS 0.35 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR H8S/2612F

```
// kernel.c
//
//
// Clock speed = 18.432 MHz
// SH7047F WS uses SCI2 for boot mode
// Kernel start address - 0xffe800

#include "iodefine.h"           // IO header file

// change following define depending on target
// #define SH
#define H8

#ifdef SH
typedef unsigned long read_datum; // unsigned long for SH
#define BLANK_VALUE                0xFFFFFFFF
#else
typedef unsigned short read_datum; // unsigned short for H8S
#define BLANK_VALUE                0xFFFF
#endif

// H8S2612F WS specific
#define FLASH_SWE        FLASH.FLMCR1.BIT.SWE
#define FLASH_PSU        FLASH.FLMCR1.BIT.PSU
#define FLASH_P           FLASH.FLMCR1.BIT.P
#define FLASH_PV         FLASH.FLMCR1.BIT.PV
#define FLASH_EBR1       FLASH.EBR1.BYTE
#define FLASH_EBR2       FLASH.EBR2.BYTE
#define FLASH_EB0        FLASH.EBR1.BIT.EB0
#define FLASH_EB1        FLASH.EBR1.BIT.EB1
#define FLASH_EB2        FLASH.EBR1.BIT.EB2
#define FLASH_EB3        FLASH.EBR1.BIT.EB3
#define FLASH_EB4        FLASH.EBR1.BIT.EB4
#define FLASH_EB5        FLASH.EBR1.BIT.EB5
#define FLASH_EB6        FLASH.EBR1.BIT.EB6
#define FLASH_EB7        FLASH.EBR1.BIT.EB7
#define FLASH_EB8        FLASH.EBR2.BIT.EB8
#define FLASH_EB9        FLASH.EBR2.BIT.EB9
#define FLASH_EB10       FLASH.EBR2.BIT.EB9
#define FLASH_EB11       FLASH.EBR2.BIT.EB9
#define FLASH_ESU        FLASH.FLMCR1.BIT.ESU
#define FLASH_E           FLASH.FLMCR1.BIT.E
#define FLASH_EV         FLASH.FLMCR1.BIT.EV

#define MAX_FLASH_ADDR          0x20000
#define FLASH_LINE_SIZE        128
#define NO_OF_FLASH_BLOCKS     10
#define XTAL                    1843200L
#define MAX_PROG_COUNT         1000
#define MAX_ERASE_ATTEMPTS     120

// 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)
```

```

const unsigned long eb_block_addr [NO_OF_FLASH_BLOCKS + 1] = {
    0x00000000L,
    0x00000400L,
    0x00000800L,
    0x00000c00L,
    0x00001000L,
    0x00008000L,
    0x0000c000L,
    0x0000e000L,
    0x00010000L,
    0x00018000L,
    0x00020000L /* 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 H8S/2612 tpu with a system clock deivider of 16
#define ONE_USEC              ((1L * XTAL) / 1600000L)
#define TWO_USEC              ((2L * XTAL) / 1600000L)
#define FOUR_USEC             ((4L * XTAL) / 1600000L)
#define FIVE_USEC             ((5L * XTAL) / 1600000L)
#define TEN_USEC              ((1L * XTAL) / 160000L)
#define TWENTY_USEC           ((2L * XTAL) / 160000L)
#define THIRTY_USEC           ((3L * XTAL) / 160000L)
#define FIFTY_USEC            ((5L * XTAL) / 160000L)
#define ONE_HUNDRED_USEC      ((1L * XTAL) / 16000L)
#define TWO_HUNDRED_USEC      ((2L * XTAL) / 16000L)
#define TEN_MSEC              ((1L * XTAL) / 1600L)

// function prototypes
void main (void);
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_035_um (unsigned char block_num);

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

volatile unsigned long delay_counter;

// Functions
unsigned char prog_flash_line_128 (unsigned long t_address, union char_rd_datum_union *p_data)
{
    // function to program one 128 byte flash line
    //
    // t_address is the start address for the flash line to be programmed

```

```

//
// 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

unsigned short n_prog_count;      // loop counter for programming attempts (0-
>MAX_PROG_COUNT)
unsigned short d;                // variable used for various loop counts
unsigned char m;                 // flag to indicate if re-programming required
(1=yes 0=no)
unsigned char ax;               // loop counter for incrementing
'uc_v_write_address' ptr
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

// enable flash writes
FLASH_SWE = 1;

// wait tSSWE
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 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 access
    dest_address = (unsigned char *) t_address;
    for (d=0; d<FLASH_LINE_SIZE; d++)
    {
        *dest_address++ = re_program_data.c[d];
    }

    // apply the write pulse
    // note that this is specified as a sub-routine call in the hw manual
    // flowchart but is part of this single function here
    //
    // if code size is a problem then placing this code in a sub-routine may be beneficial
    //
    // enter program setup
    FLASH_PSU = 1;

    // wait tSPSU
    delay (FIFTY_USEC);

    // start programming pulse
    FLASH_P = 1;
}

```

```

if (n_prog_count < 6)
    delay (THIRTY_USEC);
else
    delay (TWO_HUNDRED_USEC);

// stop programming
FLASH_P = 0;

// wait tCP
delay (FIVE_USEC);

// exit program setup
FLASH_PSU = 0;

// wait tCPSU
delay (FIVE_USEC);

// verify the data via long word 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
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;

    // wait tSPVR
    delay (TWO_USEC);

    // 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;
    }

    // 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++;
} // end of verify loop

// exit program verify mode
FLASH_PV = 0;

// wait tCPV
delay (TWO_USEC);

// 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];
    }

    // enter program setup
    FLASH_PSU = 1;

    // wait SPSU
    delay (FIFTY_USEC);

    // start programming pulse
    FLASH_P = 1;

    // wait tSP
    delay (TEN_USEC);

    // stop programming
    FLASH_P = 0;

    // wait
    delay (FIVE_USEC);

    // exit program setup
    FLASH_PSU = 0;

    // wait tCPSU
    delay (FIVE_USEC);
}

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

```

```

        // wait tCSWE
        delay (ONE_HUNDRED_USEC);

        // end of successful programming
        return (PROG_PASS);
    }

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

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

// wait tCSWE
delay (ONE_HUNDRED_USEC);

// end of failed programming
return (PROG_FAIL);
}

unsigned char erase_block_035_um (unsigned char block_num)
{
    unsigned char erase;          // flag showing erase status - BLANK or NOT_BLANK
    unsigned char ax;            // loop counter
    unsigned long attempts;      // loop counter for erase attempts (0-
>MAX_ERASE_ATTEMPTS)
    read_datum *ul_v_read;       // pointer for reading verify data
    unsigned char *uc_v_write;   // pointer for writing to verify data area

    // 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 flash writes
        FLASH_SWE = 1;

        // wait tSSWE
        delay (ONE_USEC);

        // set the correct EB bit in correct EBR register
        // this is usually device specific
        FLASH_EBR1 = 0;
        FLASH_EBR2 = 0;
        switch (block_num)
        {
            case 0:

```

```

        FLASH_EB0 = 1;
    break;

    case 1:
        FLASH_EB1 = 1;
    break;

    case 2:
        FLASH_EB2 = 1;
    break;

    case 3:
        FLASH_EB3 = 1;
    break;

    case 4:
        FLASH_EB4 = 1;
    break;

    case 5:
        FLASH_EB5 = 1;
    break;

    case 6:
        FLASH_EB6 = 1;
    break;

    case 7:
        FLASH_EB7 = 1;
    break;

    case 8:
        FLASH_EB8 = 1;           // note the change to EBR2 here!
    break;

    case 9:
        FLASH_EB9 = 1;
    break;

    case 10:
        FLASH_EB10 = 1;
    break;

    case 11:
        FLASH_EB11 = 1;
    break;
}

// initialise the attempts counter
attempts = 0;
erase = NOT_BLANK;
while ( (attempts < MAX_ERASE_ATTEMPTS) && (erase == NOT_BLANK) )
{
    // increment the attempts counter
    attempts++;

    // enter erase mode
    FLASH_ESU = 1;
}

```

```

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

// start erasing
FLASH_E = 1;

// wait tSE
delay (TEN_MSEC);

// stop erasing
FLASH_E = 0;

// wait tCE
delay (TEN_USEC);

// exit erase mode
FLASH_ESU = 0;

// wait tCESU
delay (TEN_USEC);

// enter erase verify mode
FLASH_EV = 1;

// wait tSEV
delay (TWENTY_USEC);

// verify flash has been erased
ul_v_read = (read_datum *) eb_block_addr [block_num];
uc_v_write = (unsigned char *) eb_block_addr [block_num];

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

    // wait tSEVR
    delay (TWO_USEC);

    if (*ul_v_read != BLANK_VALUE)
    {
        // this word is not erased yet
        erase = NOT_BLANK;
    }
    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
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
delay (ONE_HUNDRED_USEC);

// check if block has been erased ok
if (erase == BLANK)
{
    // successfully erased
    return ERASE_PASS;
}
else
{
    // failed to erase this block
    return ERASE_FAIL;
}
}
}

```

```

void init_delay_timer (void)
{
    TPUI.TCR.BIT.CCLR = 1; // TCNT cleared by TGRA C/M, I/C
    TPUI.TCR.BIT.CKEG = 0; // Count at rising edge
    TPUI.TCR.BIT.TPSC = 2; // Timer pre-scaler = clk / 16

    TPUI.TMDR.BIT.MD = 0; // Normal operation

    TPUI.TIOR.BIT.IOB = 0; // Output disabled
    TPUI.TIOR.BIT.IOA = 0; // Output disabled

    TPUI.TIER.BIT.TTGE = 0; // ADC start request disabled
    TPUI.TIER.BIT.TCIEU = 0; // Underflow interrupt request disabled
    TPUI.TIER.BIT.TCIEV = 0; // Overflow interrupt request disabled
    TPUI.TIER.BIT.TGIEB = 0; // TGRB interrupt request disabled
    TPUI.TIER.BIT.TGIEA = 0; // TGRA interrupt request enabled

    TPUI.TCNT = 0;
    TPUI.TGRA = 0;
    TPUI.TGRA = 0;
    TPUI.TGRB = 0;
    TPUI.TGRB = 0;
}

```

```
void delay (unsigned short d)
```

```
{
    TPU1.TSR.BIT.TGFA = 0;
    TPU1.TGRA = d;                // set compare value
    TPU1.TCNT = 0;                // clear TCNT to 0
    TPU.TSTR.BIT.CST1 = 1;       // start timer
    while(TPU1.TSR.BIT.TGFA == 0); // wait until compare value is met
    TPU.TSTR.BIT.CST1 = 0;       // stop timer
}

void main (void)
{
    init_delay_timer();

    while(1)
    {
    }
}
```

APPENDIX D – RENESAS 0.35 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR SH7047F

```
// kernel.c
//
//
// Clock speed = 44.236MHz

#include "io7047f_ws.h" // IO header file

// change following define depending on target
#define SH
// #define H8

#ifdef SH
typedef unsigned long read_datum; // unsigned long for SH
#define BLANK_VALUE 0xFFFFFFFF
#else
typedef unsigned short read_datum; // unsigned short for H8S
#define BLANK_VALUE 0xFFFF
#endif

#define FLASH_SWE FLASH.FLMCR1.BIT.SWE
#define FLASH_PSU FLASH.FLMCR1.BIT.PSU
#define FLASH_P FLASH.FLMCR1.BIT.P
#define FLASH_PV FLASH.FLMCR1.BIT.PV
#define FLASH_EBR1 FLASH.EBR1.BYTE
#define FLASH_EBR2 FLASH.EBR2.BYTE
#define FLASH_EB0 FLASH.EBR1.BIT.EB0
#define FLASH_EB1 FLASH.EBR1.BIT.EB1
#define FLASH_EB2 FLASH.EBR1.BIT.EB2
#define FLASH_EB3 FLASH.EBR1.BIT.EB3
#define FLASH_EB4 FLASH.EBR1.BIT.EB4
#define FLASH_EB5 FLASH.EBR1.BIT.EB5
#define FLASH_EB6 FLASH.EBR1.BIT.EB6
#define FLASH_EB7 FLASH.EBR1.BIT.EB7
#define FLASH_EB8 FLASH.EBR2.BIT.EB8
#define FLASH_EB9 FLASH.EBR2.BIT.EB9
#define FLASH_EB10 FLASH.EBR2.BIT.EB10
#define FLASH_EB11 FLASH.EBR2.BIT.EB11
#define FLASH_ESU FLASH.FLMCR1.BIT.ESU
#define FLASH_E FLASH.FLMCR1.BIT.E
#define FLASH_EV FLASH.FLMCR1.BIT.EV

// SH7047F WS specific
#define MAX_FLASH_ADDR 0x40000
#define FLASH_LINE_SIZE 128
#define NO_OF_FLASH_BLOCKS 12
// #define XTAL 36864000L
#define XTAL 44236800L
#define MAX_PROG_COUNT 1000
// #define BAUD_115200 9
#define BAUD_115200 11
#define MAX_ERASE_ATTEMPTS 120
```

```

// 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)
const unsigned long eb_block_addr [NO_OF_FLASH_BLOCKS + 1] = {
    0x00000000L,
    0x00001000L,
    0x00002000L,
    0x00003000L,
    0x00004000L,
    0x00005000L,
    0x00006000L,
    0x00007000L,
    0x00008000L,
    0x00010000L,
    0x00020000L,
    0x00030000L,
    0x00040000L          /* 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 SH7047F CMT with a system clock divider of 8
#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)

// function prototypes
void main (void);
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_035_um (unsigned char block_num);

// 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;

// Functions
unsigned char prog_flash_line_128 (unsigned long t_address, union char_rd_datum_union *p_data)

```



```

{
    // function to program one 128 byte flash line
    //
    // t_address is the start address for the flash line to be programmed
    //
    // 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

    unsigned short n_prog_count; // loop counter for programming attempts (0->MAX_PROG_COUNT)
    unsigned short d;           // variable used for various loop counts
    unsigned char m;           // flag to indicate if re-programming required (1=yes 0=no)
    unsigned char ax;          // loop counter for incrementing 'uc_v_write_address' ptr
    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

    // enable flash writes
    FLASH_SWE = 1;

    // wait tSSWE
    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 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 access
        dest_address = (unsigned char *) t_address;
        for (d=0; d<FLASH_LINE_SIZE; d++)
        {
            *dest_address++ = re_program_data.c[d];
        }

        // apply the write pulse
        // note that this is specified as a sub-routine call in the hw manual
        // flowchart but is part of this single function here
        //
        // if code size is a problem then placing this code in a sub-routine may be beneficial
        //
        // enter program setup
        FLASH_PSU = 1;

        // wait tSPSU
        delay (FIFTY_USEC);

        // start programming pulse
        FLASH_P = 1;
    }
}

```

```

if (n_prog_count < 6)
    delay (THIRTY_USEC);
else
    delay (TWO_HUNDRED_USEC);

// stop programming
FLASH_P = 0;

// wait tCP
delay (FIVE_USEC);

// exit program setup
FLASH_PSU = 0;

// wait tCPSU
delay (FIVE_USEC);

// verify the data via long word 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
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;

    // wait tSPVR
    delay (TWO_USEC);

    // 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;
    }

    // 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++;
} // end of verify loop

// exit program verify mode
FLASH_PV = 0;

// wait tCPV
delay (TWO_USEC);

// 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];
    }

    // enter program setup
    FLASH_PSU = 1;

    // wait SPSU
    delay (FIFTY_USEC);

    // start programming pulse
    FLASH_P = 1;

    // wait tSP
    delay (TEN_USEC);

    // stop programming
    FLASH_P = 0;

    // wait
    delay (FIVE_USEC);

    // exit program setup
    FLASH_PSU = 0;

    // wait tCPSU
    delay (FIVE_USEC);
}

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

```

```

        FLASH_SWE = 0;

        // wait tCSWE
        delay (ONE_HUNDRED_USEC);

        // end of successful programming
        return (PROG_PASS);
    }

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

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

// wait tCSWE
delay (ONE_HUNDRED_USEC);

// end of failed programming
return (PROG_FAIL);
}

unsigned char erase_block_035_um (unsigned char block_num)
{
    unsigned char erase;          // flag showing erase status - BLANK or NOT_BLANK
    unsigned char ax;            // loop counter
    unsigned long attempts;      // loop counter for erase attempts (0->MAX_ERASE_ATTEMPTS)
    read_datum *ul_v_read;       // pointer for reading verify data
    unsigned char *uc_v_write;    // pointer for writing to verify data area

    // 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 flash writes
        FLASH_SWE = 1;

        // wait tSSWE
        delay (ONE_USEC);

        // set the correct EB bit in correct EBR register
        // this is usually device specific
        FLASH_EBR1 = 0;
        FLASH_EBR2 = 0;
        switch (block_num)
        {
            case 0:
                FLASH_EB0 = 1;

```

```

        break;

    case 1:
        FLASH_EB1 = 1;
        break;

    case 2:
        FLASH_EB2 = 1;
        break;

    case 3:
        FLASH_EB3 = 1;
        break;

    case 4:
        FLASH_EB4 = 1;
        break;

    case 5:
        FLASH_EB5 = 1;
        break;

    case 6:
        FLASH_EB6 = 1;
        break;

    case 7:
        FLASH_EB7 = 1;
        break;

    case 8:
        FLASH_EB8 = 1;           // note the change to EBR2 here!
        break;

    case 9:
        FLASH_EB9 = 1;
        break;

    case 10:
        FLASH_EB10 = 1;
        break;

    case 11:
        FLASH_EB11 = 1;
        break;
}

// initialise the attempts counter
attempts = 0;
erase = NOT_BLANK;
while ( ( attempts < MAX_ERASE_ATTEMPTS ) && ( erase == NOT_BLANK ) )
{
    // increment the attempts counter
    attempts++;

    // enter erase mode
    FLASH_ESU = 1;
}

```

```

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

// start erasing
FLASH_E = 1;

// wait tSE
delay (TEN_MSEC);

// stop erasing
FLASH_E = 0;

// wait tCE
delay (TEN_USEC);

// exit erase mode
FLASH_ESU = 0;

// wait tCESU
delay (TEN_USEC);

// enter erase verify mode
FLASH_EV = 1;

// wait tSEV
delay (TWENTY_USEC);

// verify flash has been erased
ul_v_read = (read_datum *) eb_block_addr [block_num];
uc_v_write = (unsigned char *) eb_block_addr [block_num];

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

    // wait tSEVR
    delay (TWO_USEC);

    if (*ul_v_read != BLANK_VALUE)
    {
        // this word is not erased yet
        erase = NOT_BLANK;
    }
    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
        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
    delay (ONE_HUNDRED_USEC);

    // check if block has been erased ok
    if (erase == BLANK)
    {
        // successfully erased
        return ERASE_PASS;
    }
    else
    {
        // failed to erase this block
        return ERASE_FAIL;
    }
}

}

void init_delay_timer (void)
{
    // initialises compare match timer (CMT) channel 0

    // enable in module stop register
    MST.MSTCR2.BIT.MSTP12 = 0;

    // stop channel 0
    CMT.CMSTR.BIT.STR = 0;

    // channel 0 compare match interrupt disabled
    CMT.CMCSR_0.BIT.CMIE = 0;

    // system clock / 8
    CMT.CMCSR_0.BIT.CKS = 0;

    // start timer
    CMT.CMSTR.BIT.STR = 1;
}

void delay (unsigned short d)
{
    // load compare match value into the constant register
    CMT.CMCOR_0 = d;

    // clear counter
    CMT.CMCNT_0 = 0;
}

```

```
// clear compare match flag
CMT.CMCSR_0.BIT.CMF = 0;

// loop until we have a compare match
while (CMT.CMCSR_0.BIT.CMF == 0);
}

void main (void)
{
    init_delay_timer();

    while(1)
    {
    }
}
```


APPENDIX E – RENESAS 0.35 μ M FLASH PROGRAM/PROGRAM VERIFY & ERASE/ERASE VERIFY ROUTINES FOR H8/3664F MICROCONTROLLER

```
// Renesas H8/3664F example flash programming and erasing routines
//
// kernel.c
//
// Clock speed = 7.3728MHz
// H8/3664F uses SCI0 for boot mode
// Kernel start address - 0xF780

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

// H8/3664F specific
#define FLASH_SWE FLASH.FLMCR1.BIT.SWE
#define FLASH_PSU FLASH.FLMCR1.BIT.PSU
#define FLASH_P FLASH.FLMCR1.BIT.P
#define FLASH_PV FLASH.FLMCR1.BIT.PV
#define FLASH_EBR1 FLASH.EBR1.BYTE
#define FLASH_ESU FLASH.FLMCR1.BIT.ESU
#define FLASH_E FLASH.FLMCR1.BIT.E
#define FLASH_EV FLASH.FLMCR1.BIT.EV
#define FLASH_FENR FLASH.FENR.BIT.FLSHE

// H8/3664F specific
#define MAX_FLASH_ADDR 0x8000
#define FLASH_LINE_SIZE 128
#define NO_OF_FLASH_BLOCKS 5
#define XTAL 7372800L
#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)
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 H8/3664F Timer W with a clock divider of 8
#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);

// 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;

// Functions
unsigned char prog_flash_line_128 (unsigned long t_address, union char_rd_datum_union *p_data)
{
    // Function to program one 128 byte flash line
    //
    // 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
    //
    // returns:
    // PROG_PASS if programming is successful
    // PROG_FAIL if programming is unsuccessful
    //
    // data must be written to the flash in byte units
    //
    // Please note that for the H8/3664F during the dummy write, setting the PSU
    // and P bits no RTS instructions 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 H8/3664F

```

```

// hardware manual version 4 or later.
// Note: This information has been omitted in hardware manuals prior to version 4.
// Please always ensure that you are using the very latest hardware manual.
// Hardware manuals can be downloaded from the Internet by following the 'products'
// link at:-
// http://www.eu.renesas.com

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

// 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
        TMRW.GRA = TWO_USEC;
        TMRW.TCNT = 0;
        TMRW.TSR.BIT.IMFA = 0;
        TMRW.TMR.BIT.CTS = 1;
        while (TMRW.TSR.BIT.IMFA == 0);
        TMRW.TMR.BIT.CTS = 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;
        }

        // 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++;
    } // end of verify loop

    // exit program verify mode
    FLASH_PV = 0;

    // wait tCPV (2 us)
    delay (TWO_USEC);

    // 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);

```

```

}

void apply_write_pulse(unsigned short prog_pulse)
{
    // this function applies the programming pulse to the flash memory
    //
    // 'prog_pulse' contains the value to be loaded into the timer general register
    // caller must ensure that this value will provide the correct length programming
    // pulse for the timer and its clock divider
    //
    // under programming can result in either a failure to program or a reduced
    // data retention period
    //
    // over programming can permanently damage the flash cells
    //
    // Please note that for the H8/3664F during the dummy write, setting the PSU
    // and P bits no RTS instructions 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 H8/3664F
    // hardware manual version 4 or later.
    // Note: This information has been omitted in hardware manuals prior to version 4.
    // Please always ensure that you are using the very latest hardware manual.
    // Hardware manuals can be downloaded from the Internet by following the 'products'
    // link at:-
    // http://www.eu.renesas.com

    // enter program setup mode
    FLASH_PSU = 1;

    // see note at beginning of function
    // no RTS allowed here so 'apply_write_pulse' function inlined
    TMRW.GRA = FIFTY_USEC;
    TMRW.TCNT = 0;
    TMRW.TSR.BIT.IMFA = 0;
    TMRW.TMR.BIT.CTS = 1;
    while (TMRW.TSR.BIT.IMFA == 0);
    TMRW.TMR.BIT.CTS = 0;

    // start programming pulse
    FLASH_P = 1;

    // see note at beginning of function
    // no RTS allowed here so 'apply_write_pulse' function inlined
    TMRW.GRA = prog_pulse;
    TMRW.TCNT = 0;
    TMRW.TSR.BIT.IMFA = 0;
    TMRW.TMR.BIT.CTS = 1;
    while (TMRW.TSR.BIT.IMFA == 0);
    TMRW.TMR.BIT.CTS = 0;

    // stop programming
    FLASH_P = 0;

    // wait tCP (5 us)
    delay (FIVE_USEC);
}

```

```

// exit program setup mode
FLASH_PSU = 0;

// wait tCPSU (5 us)
delay (FIVE_USEC);
}

unsigned char erase_block (unsigned char block_num)
{
// This function attempts to erase the flash memory block specified by
// 'block_num' (0 -> NO_OF_FLASH_BLOCKS)
//
// returns:
// ERASE_PASS is attempt is successful
// ERASE_FAIL is attempt fails
//
// Please note that for the H8/3664F during the dummy write, setting the PSU
// and P bits no RTS instructions 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 H8/3664F
// hardware manual version 4 or later.
// Note: This information has been omitted in hardware manuals prior to version 4.
// Please always ensure that you are using the very latest hardware manual.
// Hardware manuals can be downloaded from the Internet by following the 'products'
// link at:-
// http://www.eu.renesas.com

unsigned char erase, ax, x;
unsigned long attempts;
read_datum *ul_v_read;
unsigned char *uc_v_write;

// 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 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
        TMRW.GRA = TWO_USEC;
        TMRW.TCNT = 0;
        TMRW.TSR.BIT.IMFA = 0;
        TMRW.TMR.BIT.CTS = 1;
        while (TMRW.TSR.BIT.IMFA == 0);
        TMRW.TMR.BIT.CTS = 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;
}
}
}

```

```
void init_delay_timer (void)
{
    // Stop Timer Count
    TMRW.TMR.BIT.CTS = 0;

    // Compare match A interrupt disabled
    TMRW.TIER.BIT.IMIEA = 0;

    // System clock / 8
    TMRW.TCR.BIT.CKS = 3;
}

void delay (unsigned short d)
{
    // load compare match value into the constant register A
    TMRW.GRA = d;

    // Clear counter
    TMRW.TCNT = 0;

    // Clear compare match flag
    TMRW.TSR.BIT.IMFA = 0;

    // Start the timerW
    TMRW.TMR.BIT.CTS = 1;

    // Loop until we have a compare match
    while (TMRW.TSR.BIT.IMFA == 0);

    // Stop the TimerW
    TMRW.TMR.BIT.CTS = 0;
}

void main(void)
{
}
```

APPENDIX F – RENESAS 0.18 μ M FLASH PROGRAMING & ERASING ROUTINES FOR H8/3069F

Flash.h

```
// flash.h

#ifndef _FLASH_H
#define _FLASH_H

#define H83069F
// #define INUSERBOOTMODE

#define FLASH_NO_ERROR 0x0000
#define FLASH_PROG_ERASE_DOWNLOAD_ERROR 0x0100
#define FLASH_INIT_ERROR 0x0200
#define FLASH_PROGRAMMING_ERROR 0x0400
#define FLASH_ERASING_ERROR 0x0800

union fl_fccs { /* FCCS */
    unsigned char BYTE; /* Byte Access */
    struct { /* Bit Access */
        unsigned char FWE :1; /* FWE */
        unsigned char :2; /* */
        unsigned char FLER :1; /* FLER */
        unsigned char :3; /* */
        unsigned char SCO :1; /* SCO */
    } BIT;
};

union fl_fpcs { /* FPCS */
    unsigned char BYTE; /* Byte Access */
    struct { /* Bit Access */
        unsigned char :7; /* */
        unsigned char PPVS :1; /* PPVS */
    } BIT;
};

union fl_fecs { /* FECS */
    unsigned char BYTE; /* Byte Access */
    struct { /* Bit Access */
        unsigned char :7; /* */
        unsigned char EPVB :1; /* EPVB */
    } BIT;
};

union fl_ramcr { /* RAMCR */
    unsigned char BYTE; /* Byte Access */
    struct { /* Bit Access */
        unsigned char :4; /* */
        unsigned char RAMS :1; /* RAMS */
        unsigned char RAM2 :1; /* RAM2 */
        unsigned char RAM1 :1; /* RAM1 */
        unsigned char RAM0 :1; /* RAM0 */
    } BIT;
};
```

```

};

#ifdef H83069F
union fl_fvacr {
    unsigned char BYTE; /* Byte Access */
    struct { /* Bit Access */
        unsigned char FVCHGE:1; /* FVCHGE */
        unsigned char :3; /* */
        unsigned char FVSEL3 :1; /* FVSEL */
        unsigned char FVSEL2 :1; /* FVSEL */
        unsigned char FVSEL1 :1; /* FVSEL */
        unsigned char FVSEL0 :1; /* FVSEL */
    } BIT;
};
#endif

// SH7058F
#ifndef H83069F
#define FLASH_FCCS (*(volatile union fl_fccs *)0xFFFFE800)
#define FLASH_FPCS (*(volatile union fl_fpcs *)0xFFFFE801)
#define FLASH_FECS (*(volatile union fl_fecs *)0xFFFFE802)
#define FLASH_FKEY (*(volatile unsigned char *)0xFFFFE804)
#define FLASH_FMATS (*(volatile unsigned char *)0xFFFFE805)
#define FLASH_FTDAR (*(volatile unsigned char *)0xFFFFE806)
#define FLASH_RAMER (*(volatile union fl_ramcr *)0xFFFFEC26)

#define FTDAR_START_ADDRESS_FFFF0000 0x00
#define FTDAR_START_ADDRESS_FFFF0800 0x01
#define FTDAR_START_ADDRESS_FFFF1000 0x02
#define FTDAR_START_ADDRESS_FFFF1800 0x03
#define FTDAR_START_ADDRESS_FFFF2000 0x04
#define FTDAR_START_ADDRESS_FFFF2800 0x05

#define FTDAR_ADDRESS 0xFFFF0800
#define CPU_CLOCK_FREQ 4000 // 40MHz
#define USER_BRANCH_DEST_ADDRESS 0 // no address
#endif

// H8/3069F
#ifdef H83069F
#define FLASH_FCCS (*(volatile union fl_fccs *)0xFEE0B0)
#define FLASH_FPCS (*(volatile union fl_fpcs *)0xFEE0B1)
#define FLASH_FECS (*(volatile union fl_fecs *)0xFEE0B2)
#define FLASH_FKEY (*(volatile unsigned char *)0xFEE0B4)
#define FLASH_FMATS (*(volatile unsigned char *)0xFEE0B5)
#define FLASH_FTDAR (*(volatile unsigned char *)0xFEE0B6)
#define FLASH_RAMER (*(volatile union fl_ramcr *)0xFEE077)
#define FLASH_FVACR (*(volatile union fl_fvacr *)0xFEE0B7)
#define FLASH_FVADRR (*(volatile unsigned char *)0xFEE0B8)
#define FLASH_FVADRE (*(volatile unsigned char *)0xFEE0B9)
#define FLASH_FVADRH (*(volatile unsigned char *)0xFEE0BA)
#define FLASH_FVADRL (*(volatile unsigned char *)0xFEE0BB)

#define FTDAR_START_ADDRESS_FFEF20 0x00
#define FTDAR_START_ADDRESS_FFDF20 0x01
#define FTDAR_START_ADDRESS_FFCF20 0x02
#define FTDAR_START_ADDRESS_FFBF20 0x03

```

```
#define FTDAR_ADDRESS                0xFFCF20
#define FTDAR_START_ADDRESS         FTDAR_START_ADDRESS_FFCF20
#define CPU_CLOCK_FREQ              2212    // 22.1184MHz
#define USER_BRANCH_DEST_ADDRESS   0        // no address
#endif

#define INIT_PROGRAM_ADDRESS        (FTDAR_ADDRESS + 32)
#define INIT_ERASE_ADDRESS          INIT_PROGRAM_ADDRESS
#define PROG_ROUTINE_ADDRESS        (FTDAR_ADDRESS + 16)
#define ERASE_ROUTINE_ADDRESS       PROG_ROUTINE_ADDRESS

// function prototypes
unsigned short Erase018FlashBlock( unsigned char );
unsigned short Program018FlashLine( unsigned long, unsigned char * );

#endif
```

program018.c

```
// program018.c

#include "flash.h"

#include <machine.h>

typedef unsigned short (*pt2Function)( unsigned long Address, unsigned char *ProgData );

#pragma section PTRTABLE
const pt2Function ptrtable[] = {
    Program018FlashLine
};
#pragma section

//
// The Renesas C/C++ SH compiler passes parameters in ER0 and ER1 with the return value in R0
// The Renesas C/C++ H8 compiler passes parameters in R4 and R5 with the return value in R0
// see the relevant documentation for further details
//
void func (unsigned long ul1, unsigned long ul2)
{
    // dummy function used to get the passed values into
    // registers ER0 and ER1 (H8)
    // registers R4 and R5 (SH)
}

// to use inline assembler with the Renesas C/C++ compiler the compiler output must be
// set to assembler source, this can cause problems when debugging
#pragma inline_asm( no_operation )
static void no_operation ( void )
{
    NOP
}

unsigned short Program018FlashLine( unsigned long Address, unsigned char *ProgData )
{
    //
    // Function to program a 0.18um flash line (128 bytes) starting at specified address
    // with the data pointed to by the specified pointer.
    //
    // Note:
    // This function along with the functions 'func' and 'no_operation' must all be
    // executed from on-chip RAM.
    // This means that these functions must be linked to internal RAM to ensure that any
    // references to absolute addresses refer to addresses in the internal RAM. Control
    // must not return to flash based code until this function has completed.
    //
    // While executing from internal RAM this function must not access any code or data
    // located in flash. This includes constant data and also library routines. For example,
    // when building for the H8/300H with the Renesas v4.0a compiler toolchain the library
    // functions '$sp_regsv$3' and '$spregld2$3' are used by these functions. Therefore,
    // these library routines must also be located in the internal RAM.
    //
    // One of the simplest ways to achieve this is to build the functions in this file as a
    // completely separate project which is linked to internal RAM. This RAM based code

```

```

// should then be stored in the flash by the project that is to use the functions. This
// project should then copy the RAM based code into RAM at runtime and execute it from there.
//
// There are a number of methods of taking the RAM based code, moving it to flash for storage
// and then moving it to RAM for execution. Some of these methods are described in apps
notes
// numbers REG05B0021-0100 & REG05B0023-0100
//
// If a separate project is not used for the functions in this file then any library calls
and
// constant data accesses are likely to access the flash memory unless an alternative
approach
// is adopted.
//
// parameters:
// -----
// Address          -      Address in flash which is where programming should start. The
caller
//
//                               must ensure that this address is on a flash line boundary
(128 byte)
// *ProgData -      Pointer to the data to be programmed into the flash
//
// returns:
// -----
// result of program request
// 0x0000          -      flash line programmed ok
// 0x01xx          -      download error
// 0x02xx          -      initialisation error
// 0x04xx          -      programing error
// where xx is the value indicating the exact nature of the error as specified in the ROM
// section of the hardware manual
//
//
volatile unsigned char *dfpr;          // pointer used to access the contents of the FT DAR
address containing
//
//                               // the pass or fail
information when downloading the erase routine to internal RAM
unsigned char fpr;                    // flash pass/fail result parameter
unsigned long fpefeq, fubra;          // variables used for passing CPU frequency, user branch
destination address
unsigned long fmpar, fmpdr;           // variables used for passing prog destination
start addr and data storage address
unsigned char (*fp) ( void );         // function pointer for calling the intialisation and
programming routines
unsigned short status;                // variable for calculating the return value for
this function

#ifdef H83069F
// if SH-2(e) set the vector base register to zero
set_vbr( (void **) 0 );
#endif

// initialise dfpr to point to first byte in download destination area
// specified by FDAR
dfpr = (unsigned char *) FT DAR_ADDRESS;

// set address where flash prog and erase routines will be loaded
// approx 2kB of RAM from this address will be unavailable to the user program

```

```

// while flash erasing is being performed
FLASH_FTDAR = FTDAR_START_ADDRESS;

// select flash programming program to be downloaded
FLASH_FPCS.BIT.PPVS = 1;    // download flash programming program to RAM
FLASH_FECS.BIT.EPVB = 0;    // do NOT download flash erasing program to RAM

// initialise contents of dfpr
// the contents of this pointer will contain the status of the request to download the
// erasing program to RAM
*dfpr = 0xff;

// start download of flash programming program
// disable software protection
FLASH_FKEY = 0xa5;
FLASH_FCCS.BIT.SCO = 1;
no_operation();
no_operation();
no_operation();
no_operation();
// enable software protection
FLASH_FKEY = 0;

// check that the download has completed successfully
// if *dfpr ==
// 0x00          - indicates download was successful
// 0xff          - indicates that there was something wrong with the FTDAR value
indicated by
//
// the TDER (bit 7) error bit in FTDAR
// bit 0 set - downloading of on-chip program failed (SF==1)
// bit 1 set - FKEY setting is abnormal (FK==1)
// bit 2 set - Download error as multi-selection or non-supported program selected
(SS==1)
if( *dfpr != 0 )
{
// the download has failed for some reason
status = (unsigned short) *dfpr;
status |= (unsigned short) FLASH_PROG_ERASE_DOWNLOAD_ERROR;

return status;
}

// set the operating frequency
// FPEFEQ value must be loaded into ER0 (H8) / R4 (SH)
// FUBRA value must be loaded into ER1 (H8) / R5 (SH)
// return value is in R0(L)
// dummy 'func' function used to ensure correct function call interface
fpfefeq = CPU_CLOCK_FREQ;
fubra = 0;    // user branch processing not required
func( fpfefeq, fubra );
// load the address of the erase intialisation routine into the function pointer
fp = (void *) INIT_PROGRAM_ADDRESS;
fpfr = fp();    // the returned value is in fpfr ( R0(L) )

// check that the initialisation was performed without errors
// if fpfr ==
// 0x00          - indicates initialisation was successful
// bit 0 set - initialisation failed (SF==1)

```



```

    // bit 1 set -      operating frequency invalid (FQ==1)
    // bit 2 set -      user branch address invalid (BR==1)
if( fpfr != 0 )
{
    // there has been an error
    status = (unsigned short) fpfr;
        status |= (unsigned short) FLASH_INIT_ERROR;

        return status;
}

// in either user mode or user boot mode only the user mat can be erased so
// if in user boot mode then the MAT should be switched from the user boot mat
// to the user mat
#ifdef INUSERBOOTMODE
// set FMATS to any value other than H'AA
FLASH_FMATS = 0;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// disable software protection
FLASH_FKEY = 0x5a;

// FMPAR (address in Flash where programming should start) should be in ER1 (H8) / R5 (SH)
// FMPDR (address of data) should be in ER0 (H8) / R4 (SH)
// result returned in R0(L)
// dummy 'func' function used to ensure correct function call interface
fmpar = Address;
fmpdr = (unsigned long) ProgData;
func( fmpdr, fmpar );
fp = (void *) PROG_ROUTINE_ADDRESS;
fpfr = fp();

// if in user boot mode then switch MAT back to the user boot MAT
#ifdef INUSERBOOTMODE
// set FMATS to H'AA
FLASH_FMATS = 0xAA;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// enable software protection
FLASH_FKEY = 0;

if( fpfr != 0 )
{
    // there has been an programming error
    status = (unsigned short) fpfr;
        status |= (unsigned short) FLASH_PROGRAMMING_ERROR;

        return status;
}
else

```

```
{  
    return (unsigned short) FLASH_NO_ERROR;  
}
```

erase018.c

```

// erase018.c

#include "flash.h"

#include <machine.h>

typedef unsigned short (*pt2Function)(unsigned char );           // function pointer for calling the
RAM based erase routine

#pragma section PTRTABLE
const pt2Function ptrtable[] = {
    Erase018FlashBlock
};
#pragma section

//
// The Renesas C/C++ SH compiler passes parameters in ER0 and ER1 with the return value in R0
// The Renesas C/C++ H8 compiler passes parameters in R4 and R5 with the return value in R0
// see the relevant documentation for further details
//
void func (unsigned long ul1, unsigned long ul2)
{
    // dummy function used to get the passed values into
    // registers ER0 and ER1 (H8)
    // registers R4 and R5 (SH)
}

// to use inline assembler with the Renesas C/C++ compiler the compiler output must be
// set to assembler source, this can cause problems when debugging
#pragma inline_asm( no_operation )
static void no_operation ( void )
{
    NOP
}

unsigned short Erase018FlashBlock( unsigned char FlashBlock )
{
    //
    // Function to erase the specified 0.18um flash erase block
    //
    // Note:
    // This function along with the functions 'func' and 'no_operation' must all be
    // executed from on-chip RAM.
    // This means that these functions must be linked to internal RAM to ensure that any
    // references to absolute addresses refer to addresses in the internal RAM. Control
    // must not return to flash based code until this function has completed.
    //
    // While executing from internal RAM this function must not access any code or data
    // located in flash. This includes constant data and also library routines. For example,
    // when building for the H8/300H with the Renesas v4.0a compiler toolchain the library
    // functions '$sp_regsv$3' and '$spregld2$3' are used by these functions. Therefore,
    // these library routines must also be located in the internal RAM.
    //
    // One of the simplest ways to achieve this is to build the functions in this file as a

```

```

// completely separate project which is linked to internal RAM. This RAM based code
// should then be stored in the flash by the project that is to use the functions. This
// project should then copy the RAM based code into RAM at runtime and execute it from there.
//
// There are a number of methods of taking the RAM based code, moving it to flash for storage
// and then moving it to RAM for execution. Some of these methods are described in apps
notes
// numbers REG05B0021-0100 & REG05B0023-0100
//
// If a separate project is not used for the functions in this file then any library calls
and
// constant data accesses are likely to access the flash memory unless an alternative
approach
// is adopted.
//
// parameters:
// -----
// FlashBlock - flash erase block to be erased, the caller should ensure that the value
is valid
//
// returns:
// -----
// result of erase request
// 0x0000 - block erased ok
// 0x01xx - download error
// 0x02xx - initialisation error
// 0x08xx - erasing error
// where xx is the value indicating the exact nature of the error as specified in the ROM
// section of the hardware manual
//
volatile unsigned char *dfpr; // pointer used to access the contents of the
FTDAR address which contains // the pass or fail
information when downloading the erase routine to internal RAM // variable to store the
unsigned char fpfr; // flash pass/fail result parameter
result of initialisation and erase routines
unsigned long fpefeq, fubra, febs; // variables used for passing CPU frequency, user branch
destination address // and flash erase block
number
unsigned char (*fp) ( void ); // function pointer for calling the intialisation
and programming routines
unsigned short status; // variable for calculating the return
value for this function

#ifndef H83069F
// if SH-2(e) set the vector base register to zero
set_vbr( (void **) 0 );
#endif

// initialise dfpr to point to first byte in download destination area
// specified by FDAR
dfpr = (unsigned char *) FTDAR_ADDRESS;

// set address where flash prog and erase routines will be loaded
// approx 2kB of RAM from this address will be unavailable to the user program

```

```

// while flash erasing is being performed
FLASH_FTDAR = FTDAR_START_ADDRESS;

// select flash erasing program to be downloaded
FLASH_FPCS.BIT.PPVS = 0;    // do NOT download flash programming program RAM
FLASH_FECS.BIT.EPVB = 1;    // download flash erasing program to RAM

// initialise contents of dfpr
// the contents of this pointer will contain the status of the request to download the
// erasing program to RAM
*dfpr = 0xff;

// start download of flash programming program
// disable software protection
FLASH_FKEY = 0xa5;
FLASH_FCCS.BIT.SCO = 1;
no_operation();
no_operation();
no_operation();
no_operation();
// enable software protection
FLASH_FKEY = 0;

// check that the download has completed successfully
// if *dfpr ==
// 0x00          - indicates download was successful
// 0xff          - indicates that there was something wrong with the FTDAR value
// indicated by
//                                the TDER (bit 7) error bit in FTDAR
// bit 0 set   - downloading of on-chip program failed (SF==1)
// bit 1 set   - FKEY setting is abnormal (FK==1)
// bit 2 set   - Download error as multi-selection or non-supported program selected
// (SF==1)
if( *dfpr != 0 )
{
    // the download has failed for some reason
    status = (unsigned short) *dfpr;
    status |= (unsigned short) FLASH_PROG_ERASE_DOWNLOAD_ERROR;

    return status;
}

// set the operating frequency
// FPEFEQ value must be loaded into ER0 (H8) / R4 (SH)
// FUBRA value must be loaded into ER1 (H8) / R5 (SH)
// return value is in R0(L)
// dummy 'func' function used to ensure correct function call interface
fpefeq = CPU_CLOCK_FREQ;
fubra = 0;    // user branch processing not required
func( fpefeq, fubra );
// load the address of the erase initialisation routine into the function pointer
fp = (void *) INIT_ERASE_ADDRESS;
fpfr = fp(); // the returned value is in fpfr ( R0(L) )

// check that the initialisation was performed without errors
// if fpfr ==
// 0x00          - indicates initialisation was successful
// bit 0 set   - initialisation failed (SF==1)

```

```

// bit 1 set -      operating frequency invalid (FQ==1)
// bit 2 set -      user branch address invalid (BR==1)
if( fpfr != 0 )
{
    // there has been an error
    status = (unsigned short) fpfr;
    status |= (unsigned short) FLASH_INIT_ERROR;

    return status;
}

// in either user mode or user boot mode only the user mat can be erased so
// if in user boot mode then the MAT should be switched from the user boot mat
// to the user mat
#ifdef INUSERBOOTMODE
// set FMATS to any value other than H'AA
FLASH_FMATS = 0;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// disable software protection
FLASH_FKEY = 0x5a;

// set the flash block to be erased
// FEBS parameter must be loaded into (E)R0 (H8) / R4 (SH)
// return value is in R0(L)
// dummy 'func' function used to ensure correct function call interface
febs = (unsigned long) FlashBlock;
func( febs, 0 );
fp = (void *) ERASE_ROUTINE_ADDRESS;
fpfr = fp();

// if in user boot mode then switch MAT back to the user boot MAT
#ifdef INUSERBOOTMODE
// set FMATS to H'AA
FLASH_FMATS = 0xAA;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// enable software protection
FLASH_FKEY = 0;

// check if block erased ok
if( fpfr != 0 )
{
    // there has been an erasing error
    status = (unsigned short) fpfr;
    status |= (unsigned short) FLASH_ERASING_ERROR;

    return status;
}
else

```

```
{  
    return (unsigned short) FLASH_NO_ERROR;  
}
```

APPENDIX G – RENESAS 0.18 μ M FLASH PROGRAMING & ERASING ROUTINES FOR SH7058F

flash.h

```
// flash.h

#ifndef _FLASH_H
#define _FLASH_H

// #define H83069F
// #define INUSERBOOTMODE

#define FLASH_NO_ERROR 0x0000
#define FLASH_PROG_ERASE_DOWNLOAD_ERROR 0x0100
#define FLASH_INIT_ERROR 0x0200
#define FLASH_PROGRAMMING_ERROR 0x0400
#define FLASH_ERASING_ERROR 0x0800

union fl_fccs {
    unsigned char BYTE; /* Byte Access */
    struct {
        unsigned char FWE :1; /* FWE */
        unsigned char :2; /* */
        unsigned char FLER :1; /* FLER */
        unsigned char :3; /* */
        unsigned char SCO :1; /* SCO */
    } BIT;
};

union fl_fpcs {
    unsigned char BYTE; /* Byte Access */
    struct {
        unsigned char :7; /* */
        unsigned char PPVS :1; /* PPVS */
    } BIT;
};

union fl_fecs {
    unsigned char BYTE; /* Byte Access */
    struct {
        unsigned char :7; /* */
        unsigned char EPVB :1; /* EPVB */
    } BIT;
};

union fl_ramcr {
    unsigned char BYTE; /* Byte Access */
    struct {
        unsigned char :4; /* */
        unsigned char RAMS :1; /* RAMS */
        unsigned char RAM2 :1; /* RAM2 */
        unsigned char RAM1 :1; /* RAM1 */
        unsigned char RAM0 :1; /* RAM0 */
    } BIT;
};
```



```

    } BIT;
};

#ifdef H83069F
union fl_fvacr {
    unsigned char BYTE; /* Byte Access */
    struct { /* Bit Access */
        unsigned char FVCHGE:1; /* FVCHGE */
        unsigned char :3; /* */
        unsigned char FVSEL3 :1; /* FVSEL */
        unsigned char FVSEL2 :1; /* FVSEL */
        unsigned char FVSEL1 :1; /* FVSEL */
        unsigned char FVSEL0 :1; /* FVSEL */
    } BIT;
};
#endif

// SH7058F
#ifndef H83069F
#define FLASH_FCCS (*(volatile union fl_fccs *)0xFFFFE800)
#define FLASH_FPCS (*(volatile union fl_fpcs *)0xFFFFE801)
#define FLASH_FECS (*(volatile union fl_fecs *)0xFFFFE802)
#define FLASH_FKEY (*(volatile unsigned char *)0xFFFFE804)
#define FLASH_FMATS (*(volatile unsigned char *)0xFFFFE805)
#define FLASH_FTDAR (*(volatile unsigned char *)0xFFFFE806)
#define FLASH_RAMER (*(volatile union fl_ramcr *)0xFFFFEC26)

#define FTDAR_START_ADDRESS_FFFF0000 0x00
#define FTDAR_START_ADDRESS_FFFF0800 0x01
#define FTDAR_START_ADDRESS_FFFF1000 0x02
#define FTDAR_START_ADDRESS_FFFF1800 0x03
#define FTDAR_START_ADDRESS_FFFF2000 0x04
#define FTDAR_START_ADDRESS_FFFF2800 0x05

#define FTDAR_ADDRESS 0xFFFF0800
#define FTDAR_START_ADDRESS FTDAR_START_ADDRESS_FFFF0800
#define CPU_CLOCK_FREQ 4000 // 40MHz
#define USER_BRANCH_DEST_ADDRESS 0 // no address
#endif

// H8/3069F
#ifdef H83069F
#define FLASH_FCCS (*(volatile union fl_fccs *)0xFEE0B0)
#define FLASH_FPCS (*(volatile union fl_fpcs *)0xFEE0B1)
#define FLASH_FECS (*(volatile union fl_fecs *)0xFEE0B2)
#define FLASH_FKEY (*(volatile unsigned char *)0xFEE0B4)
#define FLASH_FMATS (*(volatile unsigned char *)0xFEE0B5)
#define FLASH_FTDAR (*(volatile unsigned char *)0xFEE0B6)
#define FLASH_RAMER (*(volatile union fl_ramcr *)0xFEE077)
#define FLASH_FVACR (*(volatile union fl_fvacr *)0xFEE0B7)
#define FLASH_FVADRR (*(volatile unsigned char *)0xFEE0B8)
#define FLASH_FVADRE (*(volatile unsigned char *)0xFEE0B9)
#define FLASH_FVADRH (*(volatile unsigned char *)0xFEE0BA)
#define FLASH_FVADRL (*(volatile unsigned char *)0xFEE0BB)

#define FTDAR_START_ADDRESS_FFEF20 0x00
#define FTDAR_START_ADDRESS_FFDF20 0x01
#define FTDAR_START_ADDRESS_FFCF20 0x02

```

```
#define FTDAR_START_ADDRESS_FF20 0x03

#define FTDAR_ADDRESS 0xFFCF20
#define FTDAR_START_ADDRESS FTDAR_START_ADDRESS_FF20
#define CPU_CLOCK_FREQ 2212 // 22.1184MHz
#define USER_BRANCH_DEST_ADDRESS 0 // no address
#endif

#define INIT_PROGRAM_ADDRESS (FTDAR_ADDRESS + 32)
#define INIT_ERASE_ADDRESS INIT_PROGRAM_ADDRESS
#define PROG_ROUTINE_ADDRESS (FTDAR_ADDRESS + 16)
#define ERASE_ROUTINE_ADDRESS PROG_ROUTINE_ADDRESS

// function prototypes
unsigned short Erase018FlashBlock( unsigned char );
unsigned short Program018FlashLine( unsigned long, unsigned char * );
#endif
```

program018.c

```

// program018.c

#include "flash.h"

#include <machine.h>

typedef unsigned short (*pt2Function)( unsigned long Address, unsigned char *ProgData );

#pragma section PTRTABLE
const pt2Function ptrtable[] = {
    Program018FlashLine
};
#pragma section

//
// The Renesas C/C++ SH compiler passes parameters in ER0 and ER1 with the return value in R0
// The Renesas C/C++ H8 compiler passes parameters in R4 and R5 with the return value in R0
// see the relevant documentation for further details
//
void func (unsigned long ul1, unsigned long ul2)
{
    // dummy function used to get the passed values into
    // registers ER0 and ER1 (H8)
    // registers R4 and R5 (SH)
}

// to use inline assembler with the Renesas C/C++ compiler the compiler output must be
// set to assembler source, this can cause problems when debugging
#pragma inline_asm( no_operation )
static void no_operation ( void )
{
    NOP
}

unsigned short Program018FlashLine( unsigned long Address, unsigned char *ProgData )
{
    //
    // Function to program a 0.18um flash line (128 bytes) starting at specified address
    // with the data pointed to by the specified pointer.
    //
    // Note:
    // This function along with the functions 'func' and 'no_operation' must all be
    // executed from on-chip RAM.
    // This means that these functions must be linked to internal RAM to ensure that any
    // references to absolute addresses refer to addresses in the internal RAM. Control
    // must not return to flash based code until this function has completed.
    //
    // While executing from internal RAM this function must not access any code or data
    // located in flash. This includes constant data and also library routines. For example,
    // when building for the H8/300H with the Renesas v4.0a compiler toolchain the library
    // functions '$sp_regsv$3' and '$spregld2$3' are used by these functions. Therefore,
    // these library routines must also be located in the internal RAM.
    //
    // One of the simplest ways to achieve this is to build the functions in this file as a

```

```

// completely separate project which is linked to internal RAM. This RAM based code
// should then be stored in the flash by the project that is to use the functions. This
// project should then copy the RAM based code into RAM at runtime and execute it from there.
//
// There are a number of methods of taking the RAM based code, moving it to flash for storage
// and then moving it to RAM for execution. Some of these methods are described in apps
notes
// numbers REG05B0021-0100 & REG05B0023-0100
//
// If a separate project is not used for the functions in this file then any library calls
and
// constant data accesses are likely to access the flash memory unless an alternative
approach
// is adopted.
//
// parameters:
// -----
// Address          -      Address in flash which is where programming should start. The
caller
//
//                               must ensure that this address is on a flash line boundary
(128 byte)
// *ProgData -      Pointer to the data to be programmed into the flash
//
// returns:
// -----
// result of program request
// 0x0000          -      flash line programmed ok
// 0x01xx          -      download error
// 0x02xx          -      initialisation error
// 0x04xx          -      programing error
// where xx is the value indicating the exact nature of the error as specified in the ROM
// section of the hardware manual
//
//
volatile unsigned char *dfpr;          // pointer used to access the contents of the FTDAR
address containing
//                               // the pass or fail
information when downloading the erase routine to internal RAM
unsigned char fpr;                    // flash pass/fail result parameter
unsigned long fpefeq, fubra;          // variables used for passing CPU frequency, user branch
destination address
unsigned long fmpdr;                  // variables used for passing prog destination
start addr and data storage address
unsigned char (*fp) ( void );         // function pointer for calling the intialisation and
programming routines
unsigned short status;                // variable for calculating the return value for
this function

#ifdef H83069F
// if SH-2(e) set the vector base register to zero
set_vbr( (void **) 0 );
#endif

// initialise dfpr to point to first byte in download destination area
// specified by FDAR
dfpr = (unsigned char *) FTDAR_ADDRESS;

// set address where flash prog and erase routines will be loaded

```

```

    // approx 2kB of RAM from this address will be unavailable to the user program
    // while flash erasing is being performed
FLASH_FTDAR = FTDAR_START_ADDRESS;

// select flash programming program to be downloaded
FLASH_FPCS.BIT.PPVS = 1;    // download flash programming program to RAM
FLASH_FECS.BIT.EPVB = 0;    // do NOT download flash erasing program to RAM

// initialise contents of dfpr
// the contents of this pointer will contain the status of the request to download the
// erasing program to RAM
*dfpr = 0xff;

// start download of flash programming program
// disable software protection
FLASH_FKEY = 0xa5;
FLASH_FCCS.BIT.SCO = 1;
no_operation();
no_operation();
no_operation();
no_operation();
// enable software protection
FLASH_FKEY = 0;

// check that the download has completed successfully
// if *dfpr ==
// 0x00          - indicates download was successful
// 0xff          - indicates that there was something wrong with the FTDAR value
indicated by
//
// the TDER (bit 7) error bit in FTDAR
// bit 0 set - downloading of on-chip program failed (SF==1)
// bit 1 set - FKEY setting is abnormal (FK==1)
// bit 2 set - Download error as multi-selection or non-supported program selected
(SS==1)
if( *dfpr != 0 )
{
    // the download has failed for some reason
    status = (unsigned short) *dfpr;
    status |= (unsigned short) FLASH_PROG_ERASE_DOWNLOAD_ERROR;

    return status;
}

// set the operating frequency
// FPEFEQ value must be loaded into ER0 (H8) / R4 (SH)
// FUBRA value must be loaded into ER1 (H8) / R5 (SH)
// return value is in R0(L)
// dummy 'func' function used to ensure correct function call interface
fpfefeq = CPU_CLOCK_FREQ;
fubra = 0;    // user branch processing not required
func( fpfefeq, fubra );
// load the address of the erase intialisation routine into the function pointer
fp = (void *) INIT_PROGRAM_ADDRESS;
fpfr = fp();    // the returned value is in fpfr ( R0(L) )

// check that the initialisation was performed without errors
// if fpfr ==
// 0x00          - indicates initialisation was successful

```

```

    // bit 0 set -      initialisation failed (SF==1)
    // bit 1 set -      operating frequency invalid (FQ==1)
    // bit 2 set -      user branch address invalid (BR==1)
if( fpfr != 0 )
{
    // there has been an error
    status = (unsigned short) fpfr;
        status |= (unsigned short) FLASH_INIT_ERROR;

        return status;
}

// in either user mode or user boot mode only the user mat can be erased so
// if in user boot mode then the MAT should be switched from the user boot mat
// to the user mat
#ifdef INUSERBOOTMODE
// set FMATS to any value other than H'AA
FLASH_FMATS = 0;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// disable software protection
FLASH_FKEY = 0x5a;

// FMPAR (address in Flash where programming should start) should be in ER1 (H8) / R5 (SH)
// FMPDR (address of data) should be in ER0 (H8) / R4 (SH)
// result returned in R0(L)
// dummy 'func' function used to ensure correct function call interface
fmpar = Address;
fmpdr = (unsigned long) ProgData;
func( fmpdr, fmpar );
fp = (void *) PROG_ROUTINE_ADDRESS;
fpfr = fp();

// if in user boot mode then switch MAT back to the user boot MAT
#ifdef INUSERBOOTMODE
// set FMATS to H'AA
FLASH_FMATS = 0xAA;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// enable software protection
FLASH_FKEY = 0;

if( fpfr != 0 )
{
    // there has been an programming error
    status = (unsigned short) fpfr;
        status |= (unsigned short) FLASH_PROGRAMMING_ERROR;

        return status;
}

```

```
else  
{  
    return (unsigned short) FLASH_NO_ERROR;  
}  
}
```

erase018.c

```
// erase018.c

#include "..\userbootmodedemo\flash.h"

// #define H83069F // defined on compiler command line
// #define INUSERBOOTMODE // defined on compiler command line

#include <machine.h>

typedef unsigned short (*pt2Function)(unsigned char ); // function pointer for calling the
RAM based erase routine

#pragma section PTRTABLE
const pt2Function ptrtable[] = {
    Erase018FlashBlock
};
#pragma section

//
// The Renesas C/C++ SH compiler passes parameters in ER0 and ER1 with the return value in R0
// The Renesas C/C++ H8 compiler passes parameters in R4 and R5 with the return value in R0
// see the relevant documentation for further details
//
void func (unsigned long ul1, unsigned long ul2)
{
    // dummy function used to get the passed values into
    // registers ER0 and ER1 (H8)
    // registers R4 and R5 (SH)
}

// to use inline assembler with the Renesas C/C++ compiler the compiler output must be
// set to assembler source, this can cause problems when debugging
#pragma inline_asm( no_operation )
static void no_operation ( void )
{
    NOP
}

unsigned short Erase018FlashBlock( unsigned char FlashBlock )
{
    //
    // Function to erase the specified 0.18um flash erase block
    //
    // Note:
    // This function along with the functions 'func' and 'no_operation' must all be
    // executed from on-chip RAM.
    // This means that these functions must be linked to internal RAM to ensure that any
    // references to absolute addresses refer to addresses in the internal RAM. Control
    // must not return to flash based code until this function has completed.
    //
    // While executing from internal RAM this function must not access any code or data
    // located in flash. This includes constant data and also library routines. For example,
    // when building for the H8/300H with the Renesas v4.0a compiler toolchain the library
    // functions '$sp_regsv$3' and '$spregld2$3' are used by these functions. Therefore,
```



```

// these library routines must also be located in the internal RAM.
//
// One of the simplest ways to achieve this is to build the functions in this file as a
// completely separate project which is linked to internal RAM. This RAM based code
// should then be stored in the flash by the project that is to use the functions. This
// project should then copy the RAM based code into RAM at runtime and execute it from there.
//
// There are a number of methods of taking the RAM based code, moving it to flash for storage
// and then moving it to RAM for execution. Some of these methods are described in apps
notes
// numbers REG05B0021-0100 & REG05B0023-0100
//
// If a separate project is not used for the functions in this file then any library calls
and
// constant data accesses are likely to access the flash memory unless an alternative
approach
// is adopted.
//
// parameters:
// -----
// FlashBlock -      flash erase block to be erased, the caller should ensure that the value
is valid
//
// returns:
// -----
// result of erase request
// 0x0000          -      block erased ok
// 0x01xx          -      download error
// 0x02xx          -      initialisation error
// 0x08xx          -      erasing error
// where xx is the value indicating the exact nature of the error as specified in the ROM
// section of the hardware manual
//
//
volatile unsigned char *dfpr;          // pointer used to access the contents of the
FTDAR address which contains
// the pass or fail
information when downloading the erase routine to internal RAM
unsigned char fpfr;                    // flash pass/fail result parameter
// variable to store the
result of initialisation and erase routines
unsigned long fpefeq, fubra, febs;    // variables used for passing CPU frequency, user branch
destination address
// and flash erase block
number
unsigned char (*fp) ( void );          // function pointer for calling the intialisation
and programming routines
unsigned short status;                 // variable for calculating the return
value for this function

#ifndef H83069F
// if SH-2(e) set the vector base register to zero
set_vbr( (void **) 0 );
#endif

// initialise dfpr to point to first byte in download destination area
// specified by FDAR
dfpr = (unsigned char *) FTDAR_ADDRESS;

```

```

// set address where flash prog and erase routines will be loaded
// approx 2kB of RAM from this address will be unavailable to the user program
// while flash erasing is being performed
FLASH_FTDAR = FTDAR_START_ADDRESS;

// select flash erasing program to be downloaded
FLASH_FPCS.BIT.PPVS = 0; // do NOT download flash programming program RAM
FLASH_FECS.BIT.EPVB = 1; // download flash erasing program to RAM

// initialise contents of dfpr
// the contents of this pointer will contain the status of the request to download the
// erasing program to RAM
*dfpr = 0xff;

// start download of flash programming program
// disable software protection
FLASH_FKEY = 0xa5;
FLASH_FCCS.BIT.SCO = 1;
no_operation();
no_operation();
no_operation();
no_operation();
// enable software protection
FLASH_FKEY = 0;

// check that the download has completed successfully
// if *dfpr ==
// 0x00 - indicates download was successful
// 0xff - indicates that there was something wrong with the FTDAR value
indicated by
// the TDER (bit 7) error bit in FTDAR
// bit 0 set - downloading of on-chip program failed (SF==1)
// bit 1 set - FKEY setting is abnormal (FK==1)
// bit 2 set - Download error as multi-selection or non-supported program selected
(SS==1)
if( *dfpr != 0 )
{
    // the download has failed for some reason
    status = (unsigned short) *dfpr;
    status |= (unsigned short) FLASH_PROG_ERASE_DOWNLOAD_ERROR;

    return status;
}

// set the operating frequency
// FPEFEQ value must be loaded into ER0 (H8) / R4 (SH)
// FUBRA value must be loaded into ER1 (H8) / R5 (SH)
// return value is in R0(L)
// dummy 'func' function used to ensure correct function call interface
fpefeq = CPU_CLOCK_FREQ;
fubra = 0; // user branch processing not required
func( fpefeq, fubra );
// load the address of the erase intialisation routine into the function pointer
fp = (void *) INIT_ERASE_ADDRESS;
fpfr = fp(); // the returned value is in fpfr ( R0(L) )

// check that the initialisation was performed without errors

```

```

// if fpfr ==
// 0x00          -      indicates initialisation was successful
// bit 0 set   -      initialisation failed (SF==1)
// bit 1 set   -      operating frequency invalid (FQ==1)
// bit 2 set   -      user branch address invalid (BR==1)
if( fpfr != 0 )
{
    // there has been an error
    status = (unsigned short) fpfr;
    status |= (unsigned short) FLASH_INIT_ERROR;

    return status;
}

// in either user mode or user boot mode only the user mat can be erased so
// if in user boot mode then the MAT should be switched from the user boot mat
// to the user mat
#ifdef INUSERBOOTMODE
// set FMATS to any value other than H'AA
FLASH_FMATS = 0;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// disable software protection
FLASH_FKEY = 0x5a;

// set the flash block to be erased
// FEBS parameter must be loaded into (E)R0 (H8) / R4 (SH)
// return value is in R0(L)
// dummy 'func' function used to ensure correct function call interface
febs = (unsigned long) FlashBlock;
func( febs, 0 );
fp = (void *) ERASE_ROUTINE_ADDRESS;
fpfr = fp();

// if in user boot mode then switch MAT back to the user boot MAT
#ifdef INUSERBOOTMODE
// set FMATS to H'AA
FLASH_FMATS = 0xAA;
no_operation();
no_operation();
no_operation();
no_operation();
#endif

// enable software protection
FLASH_FKEY = 0;

// check if block erased ok
if( fpfr != 0 )
{
    // there has been an erasing error
    status = (unsigned short) fpfr;
    status |= (unsigned short) FLASH_ERASING_ERROR;
}

```

```
        return status;
    }
    else
    {
        return (unsigned short) FLASH_NO_ERROR;
    }
}
```

Website and Support

Renesas Technology Website

<http://www.renesas.com/>

Inquiries

<http://www.renesas.com/inquiry>

csc@renesas.com

All trademarks and registered trademarks are the property of their respective owners.

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (<http://www.renesas.com>)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
 - (1) artificial life support devices or systems
 - (2) surgical implantations
 - (3) healthcare intervention (e.g., excision, administration of medication, etc.)
 - (4) any other purposes that pose a direct threat to human life
 Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.