

MOTOROLA
SEMICONDUCTOR
APPLICATION NOTE

MC68F333 Flash EEPROM Programming Utilities

By Mark Maiolani and Mark Weidner

INTRODUCTION

The MC68F333 modular microcontroller (MCU) is a member of the M68300 product family. The MCU module complement includes a CPU32 processor, a single-chip integration module (SCIM), an 8-channel, 10-bit analog to digital converter (ADC), a time processor unit (TPU), a queued serial module (QSM), a 512-byte standby RAM (SRAM), a 3.5 kbyte RAM with TPU emulation capabilities (TPURAM), and two flash EEPROM modules (FLASH), one with a 16 kbyte array and the other with a 48 kbyte array.

This application note specifically describes software utilities that program and erase the FLASH modules in the MC68F333, but also gives general information that applies to other Motorola modular microcontrollers that incorporate flash EEPROM modules. Since the software utilities are device-specific, code must be modified for other members of the M68300 family, and re-written for devices in the M68HC16 family. Refer to the device user's manual for complete information, including timing and voltage parameters.

The programming and erasure software utilities are drivers for the CPU32 background debugger program, BD32. Use of BD32 allows a simple PC interface to be supported without an excessive increase in code size, and permits the MCU to be programmed with only an external programming voltage source. Because the MC68F333 has 4 kbytes of on-board RAM, there is no requirement for external memory to run the programming utilities.

Source files for routines discussed in this note are available from Motorola Freeware Data Systems. The Freeware BBS can be accessed by modem at (512) 891-3733. For Internet access via telnet/FTP, use freeware.aus.sps.mot.com. For World Wide Web access, use <http://freeware.aus.sps.mot.com/>.

THE FLASH EEPROM MODULE

Flash EEPROM provides high-density non-volatile memory that can be used for program or data storage. Each FLASH module consists of a control-register block that occupies a fixed position in MCU address space and a relocatable EEPROM array.

The control register block is shown in **Table 1**. It contains all of the registers to control mapping, timing, programming, and erasing of the array. Many of the control register bits have associated 'shadow' flash EEPROM bits. Shadow bits allow customization of the reset status of the module. For example, a module can be programmed to supply reset vectors from flash EEPROM bootstrap words. Several interlocks are included in the module to prevent accidental changes of critical parameters.

Unlike the control register block, the flash EEPROM array is not fixed to a particular memory address, but can be programmed to a particular address defined by the base address registers FEEBAH and FEEBAL. Array base addresses boundaries are typically determined by array size. For instance, a 16 kbyte array can be located at any 16 kbyte boundary in the address map. For M68300 family devices, arrays can also be configured to reside in both program and data space or in program space alone.



A flash EEPROM array can be read as either bytes, words, or long-words. FLASH modules respond to back-to-back IMB accesses, providing two-bus-cycle (four system clock) access for aligned long words. Each module can also be programmed to insert up to two wait states per access, to accommodate migration from slower external development memory without re-timing the system.

Because an array can be mapped to a number of different base addresses, it is possible for addresses in the array to overlap the addresses of its own register block or addresses used by other MCU modules, including memory that the program/erase utility is executing from. The resulting conflicts can cause programming or erasure to fail. Thus, the user must take special care to verify the array base address before attempting programming or erasure.

Programming is by byte or aligned word only, and FLASH modules support only bulk erasure. Hardware interlocks protect stored data from corruption if program/erase voltage is enabled accidentally.

Flash EEPROM Registers

Each control block contains five registers: the flash EEPROM module configuration register (FEEMCR), the flash EEPROM test register (FEETST), the flash EEPROM array base address registers (FEEBAH and FEEBAL), and the flash EEPROM control register (FEECTL). Four additional flash EEPROM words in the control block can contain bootstrap information for use during reset.

Table 1 Flash EEPROM Address Map

Access	Address	Register
S	\$YFF##0	Flash EEPROM Module Configuration (FEEMCR)
S	\$YFF##2	Flash EEPROM Test Register (FEETST)
S	\$YFF##4	Flash EEPROM Base Address High (FEEBAH)
S	\$YFF##6	Flash EEPROM Base Address Low (FEEBAL)
S	\$YFF##8	Flash EEPROM Control Register (FEECTL)
S	\$YFF##A	RESERVED
S	\$YFF##C	RESERVED
S	\$YFF##E	RESERVED
S	\$YFF##0	Flash EEPROM Bootstrap Word 0 (FEEBS0)
S	\$YFF##2	Flash EEPROM Bootstrap Word 1 (FEEBS1)
S	\$YFF##4	Flash EEPROM Bootstrap Word 2 (FEEBS2)
S	\$YFF##6	Flash EEPROM Bootstrap Word 3 (FEEBS3)
S	\$YFF##8	RESERVED
S	\$YFF##A	RESERVED
S	\$YFF##C	RESERVED
S	\$YFF##E	RESERVED

In the address map, Y = M111, where M represents the state of the MODMAP (MM) bit in the system integration module configuration register. MM defines the MSB (ADDR23) of the IMB address for MCU module. MM can be written only once after reset. An "S" in the access column indicates registers are located in supervisor data space. In M68300 family devices, access to supervisor space can be restricted, but M68HC16 devices operate only in supervisor space—see the respective CPU reference manuals for more information.

A number of control register bits have associated bits in shadow registers. The values of the shadow bits determine the reset states of the control register bits. In subsequent register diagrams, bits with reset states determined by shadow bits are shaded, and the reset state is annotated "SB". Shadow registers are programmed or erased in the same manner as a location in the array, using the address of the corresponding control registers. When a shadow register is programmed, the data is not written to the corresponding control register — the new data is not copied into the control register until the next reset. The contents of shadow registers are erased when the array is erased.

Configuration information is specified and programmed independently of the array. After reset, registers in the control block that contain writable bits can be modified. Writes to these registers do not affect the associated shadow register. Certain registers can be written only when the LOCK bit in the FEEMCR is disabled or when the STOP bit in the FEEMCR is set.

Module Configuration Register

FLASH module configuration registers (FEEMCR) control module configuration. This register can be written only when the control block is not write-locked (when LOCK = 0). All active fields and bits in the MCR take values from the associated shadow register during reset.

FEEMCR — Flash EEPROM Module Configuration Register

\$YFF##0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STOP	FRZ	0	BOOT	LOCK	0	ASPC		WAIT		0	0	0	0	0	0
RESET:															
SB	0	0	SB	SB	0	SB		SB		0	0	0	0	0	0

STOP — Stop Mode Control

0 = Normal operation

1 = Low-power stop operation

Setting the STOP bit places the module in low-power stop mode. The EEPROM array is inaccessible during low-power stop. The array can be re-enabled by clearing STOP. If STOP is set during programming or erasing, program/erase voltage is automatically turned off. However, when this is done, the enable programming/erase bit (ENPE) in the FEECTL remains set. Unless ENPE is cleared, program/erase voltage is automatically reapplied when STOP is cleared.

Since the default state of the STOP bit out of reset is determined by the value stored in the shadow MCR, it is possible for the module to come out of reset in low-power mode. The reset state of the STOP bit can also be affected by reset mode selection. Refer to the integration module section of the appropriate device user's manual for more information.

FRZ — Freeze Mode Control

0 = Disable program/erase voltage while FREEZE is asserted

1 = Allow ENPE bit to turn on the program/erase voltage while FREEZE signal is asserted

FRZ determines the response of the FLASH module to assertion of the FREEZE signal by the CPU. When FRZ = 0, the program/erase voltage is disabled while FREEZE is asserted. When FRZ = 1, the ENPE bit in the FEECTL can turn on the program/erase voltage while FREEZE is asserted.

BOOT — Boot Control

0 =Flash EEPROM module responds to the bootstrap addresses after reset

1 =Flash EEPROM module does not respond to the bootstrap addresses after reset

On reset, the BOOT bit takes on the default value stored in the shadow MCR. If BOOT = 0 and STOP = 0, the module responds to program space accesses to IMB addresses \$000000 to \$000006 following reset, and the contents of FEEBS[3:0] are used as bootstrap vectors. After address \$000006 is read, the module responds normally to control block or array addresses only.

LOCK — Lock Registers

0 = Write-locking disabled

1 = Write-locked registers protected

When LOCK is set, writes to locked registers in the control block have no effect. Once set, LOCK cannot be cleared until reset occurs. The default state of the LOCK bit out of reset is determined by the value stored in the shadow MCR. If the default state is zero, LOCK can be set once to protect the registers after initialization. Once set, LOCK cannot be cleared again until another reset occurs. When a default reset state of zero is used, the initialization routine should set LOCK to prevent inadvertent reconfiguration of the FLASH module.

ASPC[1:0] — Flash EEPROM Array Space

ASPC assigns the array to a particular address space. The default state of the ASPC field out of reset is determined by the value stored in the shadow MCR. The field can be written only when LOCK = 0 and STOP = 1. The four possible encodings for ASPC are summarized in **Table 2**. In CPU-16-based systems, only encodings for supervisor space are valid.

Table 2 Array Space Encoding

ASPC[1:0]	Type of Access
00	Unrestricted program and data space
01	Unrestricted program space
10	Supervisor program and data space
11	Supervisor program space

WAIT[1:0] — Wait States

The default state of the WAIT field out of reset is determined by the value stored in the shadow MCR. WAIT[1:0] specifies the number of wait states inserted during accesses to the FLASH module. A wait state has the duration of one system clock cycle. WAIT[1:0] affects both control block and array accesses, and can be written only if LOCK = 0 and STOP = 1. **Table 3** shows wait state encodings and corresponding clock cycles per transfer.

Table 3 Wait State Encoding

WAIT[1:0]	Wait States	Clocks/Transfer
00	0	3
01	1	4
10	2	5
11	-1	2

The value of WAIT[1:0] is compatible with the lower two bits of the $\overline{\text{DSACK}}$ field in the integration module chip-select option registers. An encoding of %11 in the WAIT field corresponds to an encoding for fast termination.

Test Register

FEETST — Flash EEPROM Test Register

\$YFF##2

This registers is used for factory test only.

Base Address Registers

The base address high register (FEEBAH) contains the 16 high-order bits of the array base address; the base address low register (FEEBAL) contains the low-order bits of the address. The number of active control bits in FEEBAL is determined by the size of the array, as shown in **Table 4**. During reset, both FEEBAH and FEEBAL take on default values programmed into associated shadow registers. After reset, if LOCK = 0 and STOP = 1, software can write to FEEBAH and FEEBAL to relocate the array.

FEEBAH — Flash EEPROM Base Address High Register

\$YFF##4

15

0



RESET:

SB

FEEBAL — Flash EEPROM Base Address Low Register

\$YFF##6

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



RESET:

SB0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Table 4 FEEBAL Bit Implementation

Array Size	Bits Used
8 Kbyte	[15:13]
Up to 16 Kbyte	[15:14]
Up to 32 Kbyte	[15]
Up to 64 Kbyte	None

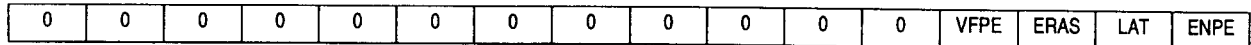
Flash EEPROM Control Register

FLASH control registers (FEECTL) control programming and erasure of the array. FEECTL is accessible in supervisor mode only. Refer to EFFECTS of LOCK Bit Operation for more information.

FEECTL — Flash EEPROM Control Register

\$YFF##8

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0



RESET:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

VFPE — Verify Program/Erase

0 = Normal read cycles

1 = Invoke program verify circuit

The VFPE bit invokes a special program-verify circuit. During programming sequences (ERAS = 0), VFPE is used in conjunction with the LAT bit to determine when programming of a location is complete. If VFPE and LAT are both set, a bit-wise exclusive-OR of the latched data with the data in the location being programmed occurs when any valid FLASH location is read. If the location is completely programmed, a value of zero is read. Any other value indicates that the location is not fully programmed. When VFPE is cleared, normal reads of valid FLASH locations occur. The value of VFPE cannot be changed while ENPE = 1.

ERAS — Erase Control

0 = Flash EEPROM configured for programming

1 = Flash EEPROM configured for erasure

The erase control bit (ERAS) in FEECTL configures the array for either programming or erasure. Setting ERAS causes all locations in the array and all control bits in the control block to be configured for erasure at the same time.

When the LAT bit is set, ERAS also determines whether a read returns the data in the addressed location (ERAS = 1) or the address itself (ERAS = 0). ERAS cannot be changed while ENPE = 1.

LAT — Latch Control

0 = Programming latches disabled

1 = Programming latches enabled

The latch control bit (LAT) in the FEECTL configures the EEPROM array for normal reads or for programming. When LAT is cleared, the FLASH module address and data buses are connected to the IMB address and data buses and the module is configured for normal reads. When LAT is set, module address and data buses are connected to parallel internal latches and the array is configured for programming or erasing.

Once LAT is set, the next write to a valid FLASH module address causes the programming circuitry to latch both address and data. Unless control register shadow bits are to be programmed, the write must be to an array address. The value of LAT cannot be changed while ENPE = 1.

ENPE — Enable Programming/Erase

0 = Disable program/erase voltage

1 = Apply program/erase voltage to flash EEPROM

Setting the enable programming/erasure (ENPE) bit in FEECTL applies program/erase voltage to the array. ENPE can be set only after LAT has been set and a write to the data and address latches has occurred. ENPE remains cleared if these conditions are not met. While ENPE is set, the LAT, VFPE, and ERAS bits cannot be changed, and attempts to read an array location are ignored.

Flash EEPROM Bootstrap Words

The bootstrap words (FEEBS[3:0]) can be used as system bootstrap vectors. When the BOOT bit in FEEMCR = 1 during reset, the FLASH module responds to program space accesses of IMB addresses \$000000 to \$000006 after reset. When BOOT = 0, the FLASH module responds only to normal array and register accesses. FEEBS[3:0] can be read at any time, but the values in the words can only be changed by programming the appropriate location. **Table 5** shows bootstrap word addresses in program space.

FEEBS[3:0] — Flash EEPROM Bootstrap Words

\$YFF##0–\$YFF##6

Table 5 Bootstrap Words

Word	Address
FEEBS0	\$00000000
FEEBS1	\$00000002
FEEBS2	\$00000004
FEEBS3	\$00000006

APPLYING FLASH PROGRAM ERASE VOLTAGE

A voltage of at least $V_{DD} - 0.5\text{ V}$ must be applied at all times to the V_{FPE} pins or damage to the FLASH module can occur. FLASH modules can be damaged by power-on and power off V_{FPE} transients. V_{FPE} must not rise to programming level while V_{DD} is below specified minimum value, and must not fall below minimum specified value while V_{DD} is applied. **Figure 1** shows the V_{FPE} and V_{DD} operating envelope.

Use of an external circuit to condition V_{FPE} is recommended. **Figure 2** shows a simple circuit that maintains required voltages and filters transients. V_{FPE} is pulled up to V_{DD} via Schottky diode D2. Application of programming voltage via diode D1 reverse-biases D2, protecting V_{DD} from excessive reverse current. D2 also protects the FLASH from damage should programming voltage go to zero. Programming power supply voltage must be adjusted to compensate for the forward-bias drop across D1. The charge time constant of R1 and C1 filters transients, while R2 provides a discharge bleed path for C1. Allow for RC charge and discharge time constants when applying and removing power. When using this circuit, keep leakage from external devices connected to the V_{FPE} pin low, to minimize diode voltage drop.

There are a number of interlocks designed to prevent accidental programming or erasure. For increased protection, raise the V_{FPE} input to programming voltage only immediately prior to issuing a PROG or BULK command, and remove programming voltage as soon as the operation is complete.

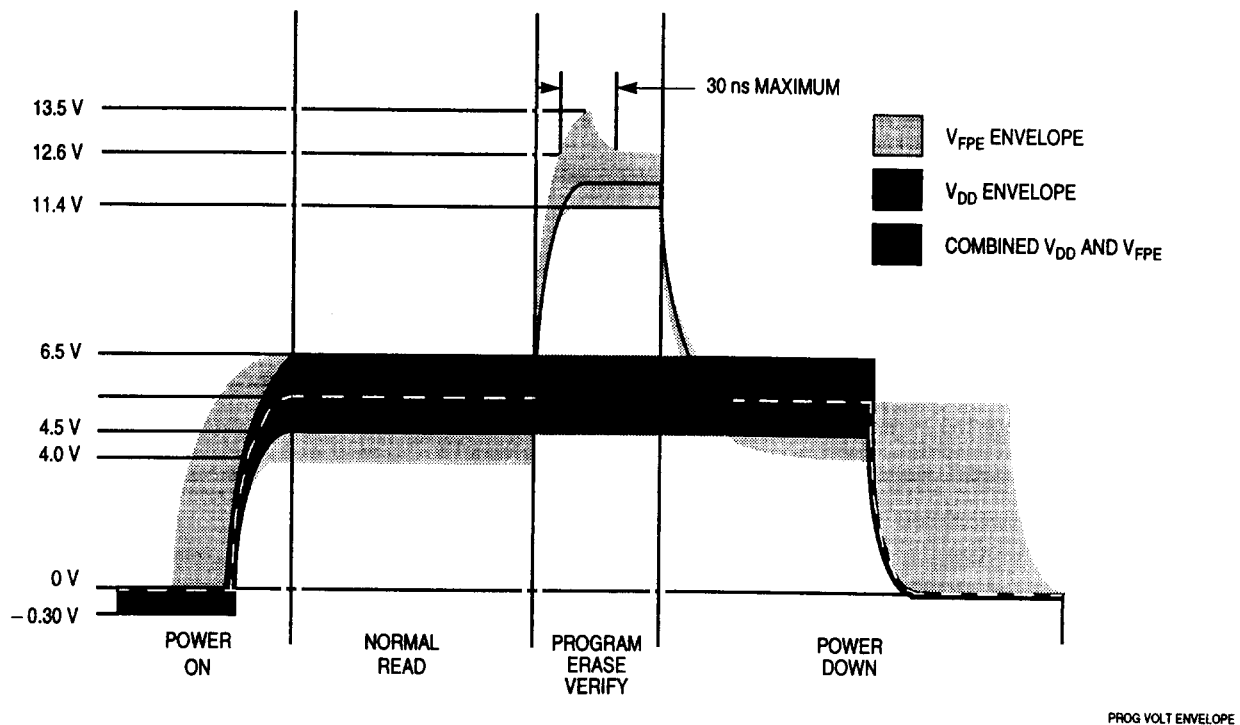


Figure 1 Programming Voltage Envelope

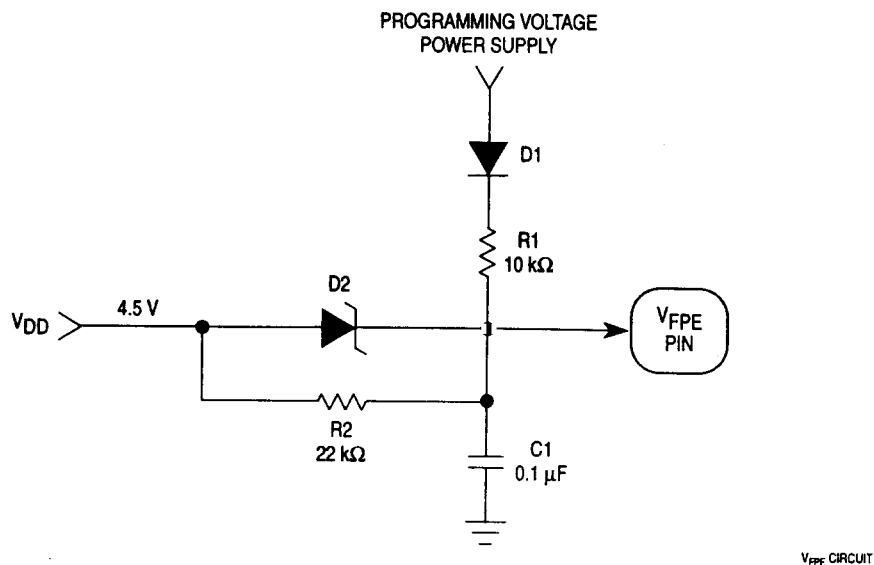


Figure 2 V_{FPE} Conditioning Circuit

EFFECTS OF LOCK BIT OPERATION

FLASH modules can be configured to prohibit access to the base address registers and the module configuration register. This capability prevents application failures caused by accidental writes to the registers. Access is controlled by the LOCK bit in the module configuration register (FEEMCR).

Because it restricts relocating the array to resolve address conflicts, the LOCK bit can also affect programming and erasing. Conflicts arise when the array is mapped to an address range that coincides with the addresses of other MCU resources. These resources may be:

1. FLASH module control register blocks
2. Control registers of other IMB modules
3. Memory required by the driver software

The third type of conflict is easily resolved by relocating the driver. BD32 macro files provide a convenient way to do this, and all other required configuration. Two example macro files, SRAMHIGH.DO and SRAMZ-ERO.DO are listed and used in the example section.

The first two conflict types require the array to be remapped. However, if the LOCK bit is set, it is not possible to immediately relocate the array by writing to the base address registers — instead, the module shadow registers must be reprogrammed so that the array will be mapped to the new address after reset.

The following procedure, also shown in Example 1, avoids possible address conflicts. It is recommended for routine programming of a blank FLASH module.

1. Program the shadow registers for the required configuration and array address
2. Reset and re-initialize the device
3. Program the array

Erasing an array which has been programmed this way should not cause problems, as the module is never in a programmed state with a conflicting array address range. If the array has been mapped to a conflicting address, it must be relocated before erasure to avoid an erase fail during the blank-check process. If the LOCK bit is clear, the array can be remapped by writing FEEBAH/L, otherwise it is necessary to perform steps 1 and 2 before erasing.

BD32 BACKGROUND DEBUGGER

BD32 is a debugger program for CPU32-based devices that executes on an IBM PC-compatible host, and communicates with the background debugging mode (BDM) port of the device via the PC printer port. Use of BDM makes a ROM-based monitor program unnecessary, and the only requirement for using it is access to the CPU32 BDM signals. If the design includes the recommended 10 pin Berg-type connector to provide access to the signals, BDM can even be used with the final application hardware.

BD32 supports a method of extending the available functions through custom driver programs. If a command that is not part of the standard command set is entered, BD32 searches the PC disk for a file with the command name and the extension.D32. If a matching file is found, it is executed by the MCU in response to the command. Parameters can be entered with the command, and are passed to the driver program as an ASCII text list in memory, pointed to by one of the processor registers.

To ensure that drivers will operate on application hardware systems with differing memory maps, BD32 requires that driver programs be relocatable, and uses a load address specified by the BD32 'driver' command. This feature is used often when programming and erasing the FLASH modules, as the drivers must not be placed in an address range which will be overwritten by a flash array.

Table 6 shows available BD32 system calls. A driver program executes these calls by executing a BGND instruction with register D0 containing the appropriate fcode value. Please refer to the BD32 documentation file BD32.DOC for more information concerning the debugger.

Table 6 BDM32 Command Summary

Name	Function	fcode	Parameters
QUIT	stop driver execution	0	None
PUTS	display character string on screen	1	A0 - address of string
PUTCHAR	display single character on screen	2	D1 - character
GETS	get string from user (CR ends)	3	A0 - address of buffer
GETCHAR	get single character from user	4	char returned in D0
GETSTAT	returns char ready/not ready status	5	D0 non-zero if ready
FOPEN	open disk file on host PC	6	A0 - filename string A1 - pointer to mode
FCLOSE	close disk file	7	D1 - file handle
FREAD	read n bytes from disk file	8	D1 - file handle D2 - byte count A0 - buffer address
FWRITE	write n bytes to disk file	9	D1 - file handle D2 - byte count A0 - buffer address
FTELL	return current file pointer pos.	10	D1 - file handle
FSEEK	seek to position n in disk file	11	D1 - file handle D2 - offset
FGETS	read \n-terminated string from file	12	D1 - file handle A0 - buffer
FPUTS	write null terminated string to file	13	D1 - file handle A0 - buffer
EVAL	evaluate expression from string	14	A0 - string D1 - return value
FREADSREC	read S-record from disk file	15	D1 - file handle A0 - buffer

PROGRAM/ERASE OPERATION

An erased bit has a logic state of one. A bit must be programmed to change its state from one to zero. Erasing a bit returns it to a logic state of one. Programming and erasing the FLASH module requires a series of control register writes and a write to an array address. The same procedure is used to program control registers that contain flash shadow bits. Programming is restricted to a single byte or aligned word at a time. The entire array and the shadow register bits are erased at the same time.

When multiple FLASH modules share a single V_{FPE} pin, do not program or erase more than one module at a time. Normal accesses to modules that are not being programmed are not affected by programming or erasure of another FLASH module.

Following paragraphs give step-by-step procedures for programming and erasure of flash EEPROM arrays. Parameters used in the descriptions are defined and characterized in the electrical specifications section of the appropriate device manual.

Programming

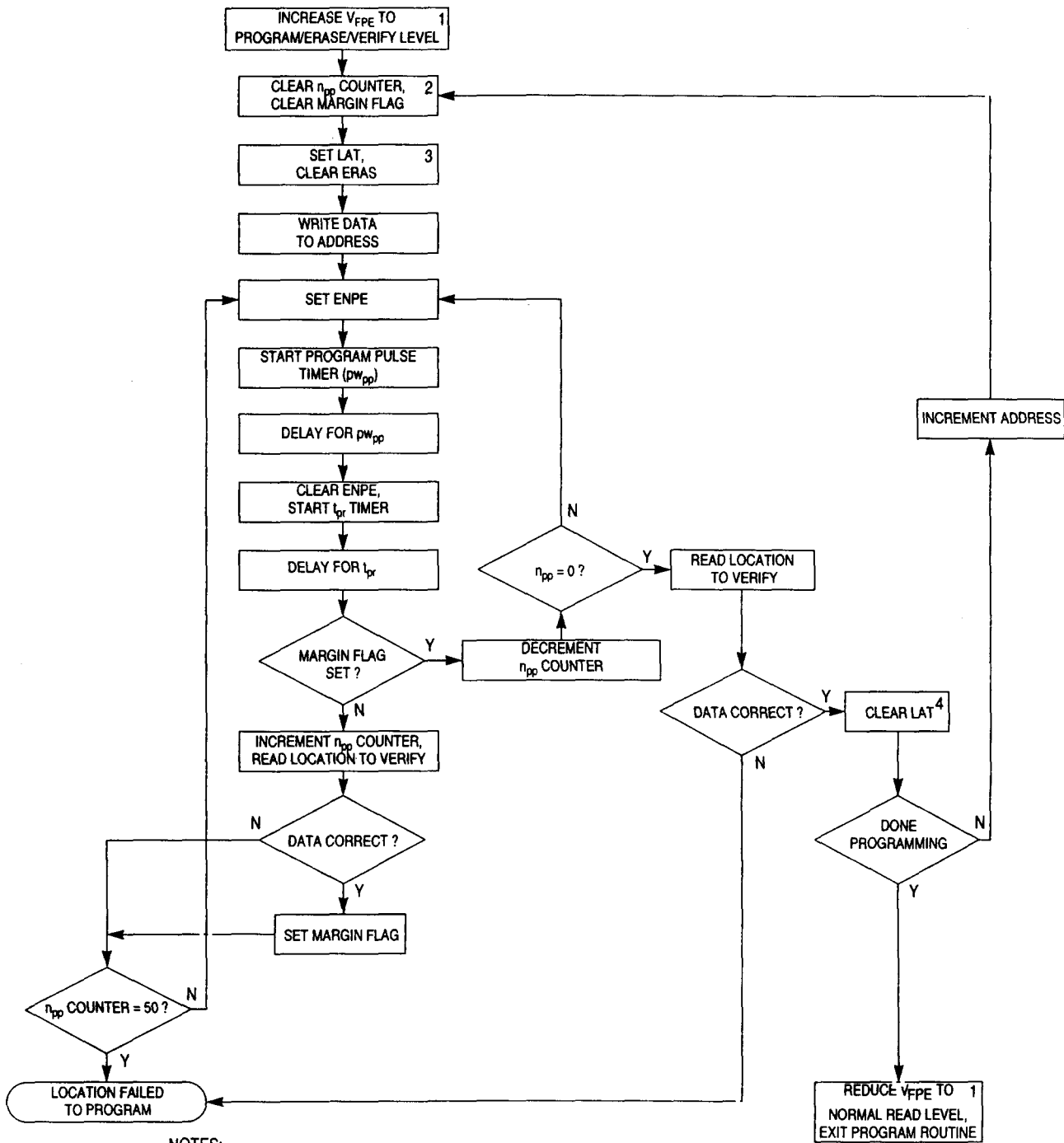
The following steps are performed to program the array. **Figure 3** is a flowchart of programming operation.

1. Increase voltage applied to the V_{FPE} pin to program/erase/verify level.
2. Clear the ERAS bit and set the LAT bit in FEECTL. This enables the programming address and data latches.
3. Write data to the address to be programmed. This latches the address to be programmed and the programming data.
4. Set the ENPE bit in FEECTL. This starts the program pulse.
5. Delay the proper amount of time for one programming pulse to take place. Delay is specified by parameter pw_{pp} .
6. Clear the ENPE bit in FEECTL. This stops the program pulse.
7. Delay while high voltage to array is turned off. Delay is specified by parameter t_{pr} .
8. Read the address to verify that it has been programmed.
9. If the location is not programmed, repeat steps 4 through 7 until the location is programmed, or until the specified maximum number of program pulses has been reached. Maximum number of pulses is specified by parameter n_{pp} .
10. If the location is programmed, repeat the same number of pulses as required to program the location. This provides 100% program margin.
11. Read the address to verify that it remains programmed.
12. Clear the LAT bit in FEECTL. This disables the programming address and data latches.
13. If more locations are to be programmed, repeat steps 2 through 10.
14. Reduce voltage applied to the V_{FPE} pin to normal read level.

Erase

The following steps are performed to erase the array. **Figure 4** is a flowchart of erasure operation.

1. Increase voltage applied to the V_{FPE} pin to program/erase/verify level.
2. Set the ERAS bit and the LAT bit in FEECTL. This configures the module for erasure.
3. Perform a write to any valid address in the control block or array. The data written does not matter.
4. Set the ENPE bit in FEECTL. This applies the erase voltage to the array.
5. Delay the proper amount of time for one erase pulse. Delay is specified by parameter t_{epk} .
6. Clear the ENPE bit in FEECTL. This turns off erase voltage to the array.
7. Delay while high voltage to array is turned off. Delay is specified by parameter t_{er} .
8. Read the entire array and control block to ensure all locations are erased.
9. If all locations are not erased, calculate a new value for t_{epk} ($t_{ej} \times \text{pulse number}$) and repeat steps 3 through 10 until all locations erase, or the maximum number of pulses has been applied.
10. If all locations are erased, calculate the erase margin (e_m) and repeat steps 3 through 10 for the single margin pulse.
11. Clear the LAT and ERAS bits in FEECTL. This allows normal access to the flash.
12. Reduce voltage applied to the V_{FPE} pin to normal read level.

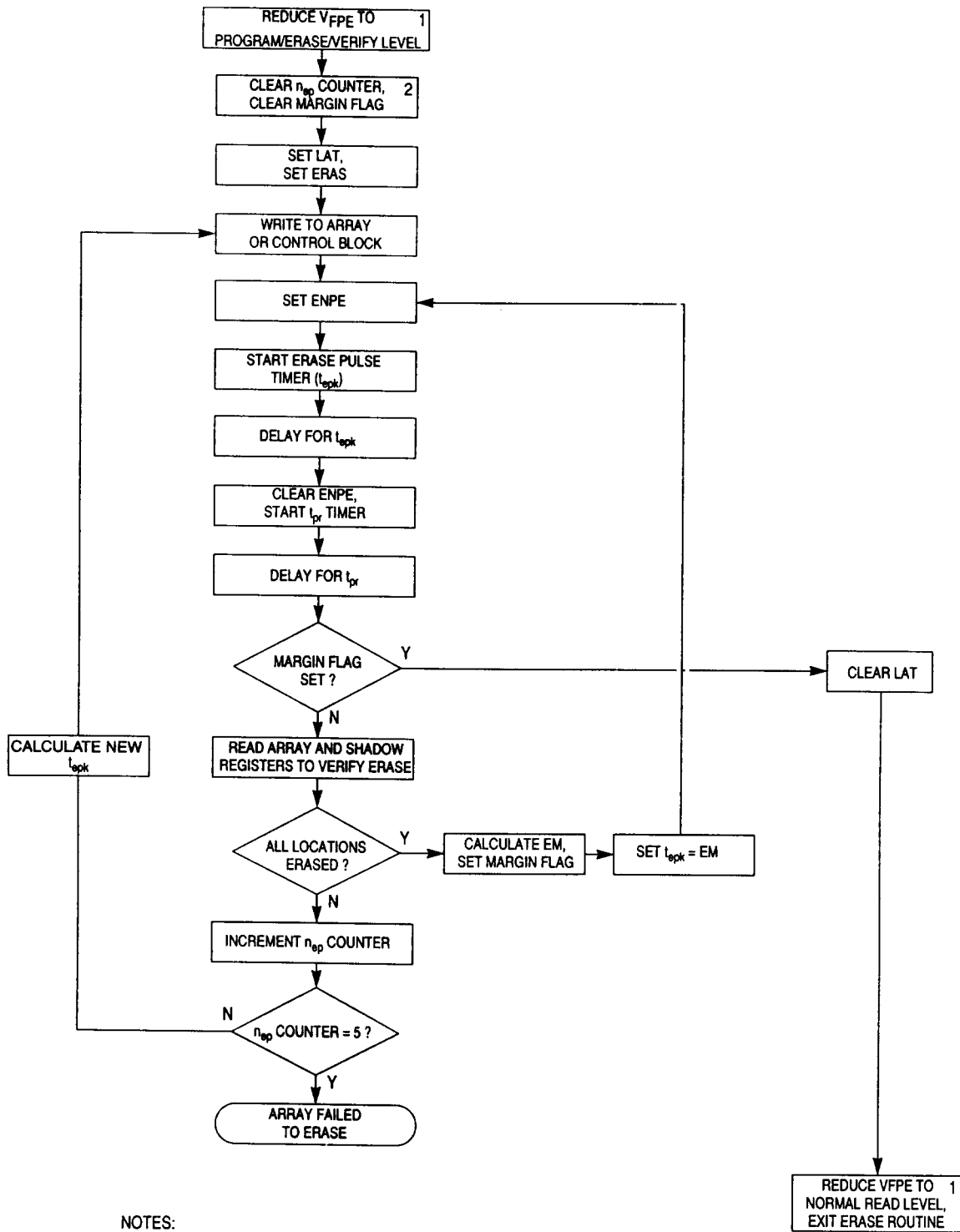


NOTES:

1. SEE ELECTRICAL CHARACTERISTICS FOR V_{FPE} PIN VOLTAGE SEQUENCING.
2. THE MARGIN FLAG IS A SOFTWARE-DEFINED FLAG THAT INDICATES WHETHER THE PROGRAM SEQUENCE IS GENERATING PROGRAM PULSES OR MARGIN PULSES.
3. TO SIMPLIFY THE PROGRAM OPERATION, THE V_{FPE} BIT IN FEE_{CTL} CAN BE SET.
4. CLEAR V_{FPE} BIT ALSO IF ROUTINE USES THIS FUNCTION.

FEEPROM PGM FLOW1.TC

Figure 3 Programming Flow



- NOTES:
1. SEE ELECTRICAL CHARACTERISTICS FOR V_{FPE} PIN VOLTAGE SEQUENCING.
 2. THE MARGIN FLAG IS A SOFTWARE-DEFINED FLAG THAT INDICATES WHETHER THE PROGRAM SEQUENCE IS GENERATING ERASE PULSES OR MARGIN PULSES.

FEEPROM PGM FLOW2.TD

Figure 4 Erasure Flow

DRIVER SOFTWARE

Driver Relocatability

Because a user can define a driver execution address to be anywhere in the MCU memory map, the BD32 driver system requires that driver code be fully relocatable. Accesses to variables that are relative to the driver location (e.g. variables within the driver area) therefore cannot use absolute addressing. Instead, use either PC-relative addressing or offset addressing using register A6. The latter is possible because BD32 writes A6 with the base address of the driver before the driver code is executed, and has the advantage of allowing writes in a single instruction. Because the CPU32 regards PC-relative addresses as non-alterable locations, an extra LEA instruction is required when writing a location using this addressing mode.

Special care is also required when accessing driver parameters as these cannot be guaranteed to be on word boundaries. Byte accesses are always used in this case to guarantee correct operation regardless of driver load address and size/number of driver parameters.

Exception Handling

Basic exception handling routines are built into the PROG and BULK drivers. In normal use no exceptions are generated, so the handlers simply indicate that an error has caused an exception. Such errors are typically caused by array address conflicts described in **EFFECTS OF LOCK BIT OPERATION**.

PROG — Flash Programming Driver

User Details

The PROG driver is designed to enable programming of flash EPROM from an S-record file on the PC running BD32. The syntax used is:

```
PROG <filename.ext> [<start address>]
```

where <filename.ext> is the filename of the S-record file, and <start address> is an optional parameter that, if specified, defines the start address of programming, overriding the start address specified in the S-record. The relative addresses of bytes in the S-record are preserved, with a fixed offset added to each S-record address. The offset is calculated as:

$$\text{offset} = (\text{start address parameter}) - (\text{first S-record address})$$

If <start address> is not specified, the addresses defined in the S-record file are used unchanged.

Each byte or word is verified after programming. Any verify errors are indicated by an error message, and the user is given the option to abort or continue programming. This facility is useful if an array is already partially programmed or damaged, or if the S-record contains programming data for a location not within any FLASH address range.

For each byte or word to be programmed, the PROG utility searches through all of the possible FLASH module addresses to find a match. PROG does not initialize the array base addresses before programming, so the user must ensure that these are correctly configured.

When specifying programming data for the shadow registers, unimplemented shadow bits must be set to zero, to avoid verify errors. Registers that may have unimplemented shadow bits are FEEMCR, FEEBAH and FEEBAL. Make certain that the array address does not overlap registers of the flash EEPROM module or another module. See **FINDING ERRORS** for more detail.

Software Details

The PROG routine applies programming pulses to the flash array until the location verifies as correctly programmed. A final series of pulses is applied for programming margin. The following sequence of steps is used to program the flash EEPROM array.

The source files for the PROG driver software are:

PROG.S62 Program code source file
PROG.MSG Message text file used by BD32
IPD.INC Definitions required for the BD32 system calls

M68F333.INC MC68F333 constants definition file, including register addresses, other flash module information, and programming/erasure timing data. Timing information is compatible with the definitions used in the MC68F333 device specification to simplify updates.

Common include files used by both drivers are shown after the erasure driver code.

PROG Driver Listing

```
*****
* 'PROG' Resident Command Driver for MC68F333 device
*
* Utility to program an MC68F333 flash EPROM module from an S record file
*
* Source file    : prog.s62
* Object file    : prog.d32
* Include files : M68F333.inc    (M68F333 addresses and programming constants)
*                ipd.inc        (BD32 system call constants)
* Message file   : prog.msg
*
* Object file format: Motorola S-records
*
* Execute as: prog <filename> [<start_address>]
* Usage : Start_address specifies start of memory to be programmed, if not
*        specified the S-record start address is used
*
* Addressing modes : This code is designed as a driver for the BD32 background
*                   debugger for CPU32 devices. A requirement is that the code must be
*                   fully relocateable. All addresses (apart from fixed module addresses)
*                   are relative, and where word alignment is not guaranteed, byte
*                   accesses must be used.
*
* Word alignment : The embedded text strings have been adjusted in size so
*                   that the following code remains word aligned - any modifications
*                   to these strings should be adjusted accordingly. An assembler
*                   'even' type directive to force word-alignment could be used if
*                   available.
*
* 32/23 bit addressing : All flash addresses are forced to 24 bits, with
*                        upper MSB ignored, so that $xxfff800 will always access FEE1MCR etc.
*****
*                    Include files
*                    lib        ipd.inc                    BD32 call code definitions
*                    lib        M68F333.inc                M68F333 device constants
*
*                    BD32 return error codes : see file PROG.MSG for associated text
UsageError        equ        1                    Usage: ...
FileError        equ        2                    Error opening file...
EvalError1       equ        3                    Error evaluating start address
EvalError2       equ        4                    Error evaluating end address
SRecError        equ        4                    Starting value for SRec errors
SRecEOFError     equ        5                    Reached EOF on input file
SRecS9Error      equ        6                    S9 read (not an error)
SRecChecksum     equ        7                    Checksum error in record
SRecFormat       equ        8                    Format error in S-record file
ProgError        equ        9                    Error programming data
ExcepError       equ       10                    Unhandled exception error
ProgOK           equ       11                    Good return value, programmed OK
```

```

*          BD32 call return codes : see bd32 file BD32.DOC
SRecS9    equ      2          ReadSRecord call - S9 Record read, end of file

*          Flash control register constants
*          FEEMCR
flashdis  equ      $90c0      Module DISABLED, disable VFPE in BDM,
*                               no boot, unrestricted space, 2 cycle access
flashen   equ      $10c0      Module ENABLED, disable VFPE in BDM,
*                               no boot, unrestricted space, 2 cycle access
*          FEECTL
latch     equ      $a          Enable prog latch
prgen     equ      $b          Enable prog volts
shadow    equ      $2          Read shadow reg
norm      equ      $0          Normal operation

*          Variable area
section  data
buffer   dc.l      Prog          start address (add load offset)
         ds.b      40          space for S-record from host
         ds.l      40          stack area
*                               initial stack pointer
stack

StartAddr ds.l      1          start address parameter

ModeAddr  dc.w      $0          address mode
OffsetAddr dc.l      $0          calculated S-record offset
FilePtr   ds.l      1          file pointer
FileName  ds.b      64          file name
Error     ds.w      1          error code

*****
*          CUSTOM VECTOR TABLE (reserved space)
*****
vectable  ds.l      13          Alternate vector table
*****
*          EXCEPTION HANDLER ROUTINE
*          Use - Quits to BD32 with unhandled exception error code
*          Exception handling is included because many user errors
*          (mapping of flash/drivers etc) could cause bus errors,
*          f-line exceptions etc. Flash programming voltage is disabled
*          in case exception occurred during a programming cycle
*****
excep_h   move.w    #norm,FEECTL(a1)  normal flash reads/writes
*                               disable programming voltage
*                               unhandled excep error
         move      #ExcepError,Error(A5)
         bra      Prog_end

FileMode  dc.b      'r',0          read mode for file open syscall
*****
*          Execution start of driver 'PROG'
*          Entry (from BD32) :
*          d0 - number of driver parameters
*          a0 - address of parameter array
*          a5 - driver offset address
*          Usage :
*          a7 - stack pointer
*****
Prog
*          ***** Exception handler initialisation
         lea.l    vectable(PC),a1    get start of vector table
         movea.l a1,a2              working (loop) copy
         lea.l    excep_h(PC),a3    get address of handler
         move.w   #$0c,d1           initialise copy loop
vecloop   move.l   a3,(a2)+         build new vector table
         dbf     d1,vecloop
         movec.l a1,vbr            set up vbr for new table

*          ***** SP and general register initialisation
         lea.l    stack(A5),a7      set up stack
*          lea.l    stack(PC),a7    set up stack (equivalent)
         move.l   a0,a2             get argv into a2
         move.l   d0,d2            get argc into d2

```

```

*          ***** Print signon and warning message
bsr      Print          print signon message
dc.b    'M68F333 Flash EEPROM Programmer Version 2.0',13,10,0

*          ***** Main initialisation
bsr      Initialize          init hardware and address list
tst     d0
bne     Prog_end

*          ***** Check command line
cmpi    #2,d2          argc < 2?
bcs     Prog_0
cmpi    #3,d2          argc > 3?
bls     Prog_1
Prog_0   move     #UsageError,Error(A5)  arg count is wrong
bra     Prog_end

*          ***** Get filename, open file, check if OK
Prog_1   addq.l  #4,a2          skip over program name
move.l  (a2)+,a0        get file name of S records
lea.l   FileMode(A5),a1  read mode - "r"
bsr     fopen
move.w  d0,FilePtr(A5)   save file pointer
bne     Prog_11         continue if OK
move    #FileError,Error(A5)  can't open input file
bra     Prog_end

*          ***** Evaluate remaining parameters
Prog_11  clr.w   ModeAddr(A5)      Assume no offset first..
cmpi    #3,d2          argv = 3 ?
bne     Prog_2
move.l  (a2)+,a0        evaluate start address parameter
bsr     Eval
beq     Prog_12
move    #EvalError1,Error(A5)
bra     Prog_3
Prog_12  move.l  d1,StartAddr(A5)   got first param
move.w  #$1,ModeAddr(A5)          signal to calculate offset

*          ***** Read an S-Record, check for errors
Prog_2   bsr     ReadsRecord        get next S Record
tst     d0
beq     Prog_25         continue if no error
cmpi    #SRecS9,d0     S9 record ?
beq     Prog_3          yes - close normally
addi    #SRecError,d0  otherwise flag error
move    d0,Error(A5)
bra     Prog_3

*          ***** Program data from S-Record into EEPROM
Prog_25  bsr     ProgRecord          program data from S Record
tst     d0
beq     Prog_2          loop till done
bsr     not_prog        print fault address
move    #ProgError,Error(A5)  error - report it

*          ***** Close input file
Prog_3   bsr     CloseInputFile      close file

*          ***** Report any errors, exit back to BD32
Prog_end move    Error(A5),d1        get error code
moveq.l #BD_QUIT,d0        exit program
bgnd

```



```

*****
* ReadSRecord - reads one S record from FilePtr
* Exit       - d0 contains returned status
*            - d1 corrupted
*            - a0 points to s-record (buffer)
*****
ReadSRecord   move.w   FilePtr(A5),d1           file pointer
              lea.l   buffer(A5),a0           point to S Record buffer
              moveq.l #BD_FREADSREC,d0
              bgnd
              rts

*****
* CloseInputFile - closes FilePtr
* Exit         - d0 corrupted
*            - d1 corrupted
*            - does not affect Error
*****
CloseInputFile move.l   FilePtr(A5),d1
              moveq.l #BD_FCLOSE,d0
              bgnd
              rts

*****
* Eval        - evaluates numeric string
* Entry      - string address in a0
* Exit       - result in D1, error flag in D0
*****
Eval         moveq.l #BD_EVAL,d0
              bgnd
              tst     d0
              rts

*****
* fopen      - performs file open routine
* Entry     - filename pointer in A0
*            - file mode pointer in A1
* Exit     - file pointer in D0
*****
fopen       moveq.l #BD_FOPEN,d0
              bgnd
              rts

*****
* FindStrEnd - searches an ASCII string for end of string
*            - marker ('null'/ 0 char)
* Entry     - string pointed to by A0
* Exit     - returns a0 pointing to end of string marker
*****
FindStrEnd  move.w   d0,-(a7)                 push temp register
              moveq  #-1,d0                   max loop count 1st time thru
FSE_1      tst.b   (a0)+                       byte == 0?
              dbeq   d0,FSE_1                 uses loop mode
              bne   FSE_1                     loop till test true
              subq.l #1,a0                     decrement address reg.
              move.w (a7)+,d0                 restore register
              rts

*****
* ntoh      - prints hex value of register D0 least sig nibble to screen
* Entry    - D0 contains nibble value
*****
ntoh       movem.l d0/d1,-(a7)
              move.b d0,d1
              andi.w #$f,d1
              addi.b #'0',d1
              cmpi.b #10+'0',d1
              bcs   nt_1
              addi.b #'A'-'9'-1,d1
nt_1      moveq  #BD_PUTCHAR,d0
              bgnd
              movem.l (a7)+,d0/d1
              rts

```

```

*****
* btoh          - prints hex value of byte register D0 to screen
* Entry         - D0 contains byte value
*****
btoh           ror.b  #4,d0
               bsr    ntoh
               ror.b  #4,d0
               bsr    ntoh
               rts

*****
* wtoh          - prints hex value of word register D0 to screen
* Entry         - D0 contains word value
*****
wtoh           ror.w  #8,d0
               bsr    btoh
               ror.w  #8,d0
               bsr    btoh
               rts

*****
* ltoh          - prints hex value of long word register D0 to screen
* Entry         - D0 contains long word value
*****
ltoh           swap   d0
               bsr    wtoh
               swap   d0
               bsr    wtoh
               rts

*****
* Print         - prints constant string in code and returns to
*               program at first even location after string
* Entry         - parameters indexed from stacked return PC
* Exit         - stacked return PC modified to give correct return
*               no registers corrupted
*****
Print          movem.l a0/d0,-(a7)          save registers
*              WARNING : Any change to movem list will require change
*              to stack offset used below
*              move.l  8(a7),a0             get address of string
*                                   ( = stacked return address)
*              moveq.l #BD_PUTS,d0         function call
*              bgnd
*              bsr    FindStrEnd           get end of ASCII string
*              move.l a0,d0                test for odd address
*              addq.l #1,d0                skip past end of string
*              btst  #0,d0
*              beq   Print_1
*              addq.l #1,d0
Print_1        move.l d0,8(a7)              it's odd - return to next addr
               movem.l (a7)+,d0/a0        update stacked return address
               rts                        get back registers
               done

*****
* crlf          - prints carriage return, line feed combo
* Exit         - no registers corrupted
*****
crlf           bsr    Print                carriage return, line feed
               dc.b  13,10,0,0
               rts

*****
* getchar       - returns character typed by user
* Exit         - d0 contains character typed
*****
getchar        moveq.l #BD_GETCHAR,d0
               bgnd
               rts

```

```

*****
* usedelay - programmable software delay loop
* Entry - delay in us (approximate) stored in d1,
* legal values are 2 ... 65535
* Exit - d1 corrupted
* Environment- timings assume 2 clock program memory access and 16.778MHz
* clkout frequency
*****
*
*          jsr      usdelay          13
usdelay   subq     #2,d1             2 - adjust for overhead
          asl     #1,d1             6 - multiply count by 2 for us
loop      tst      d1                2
          dbf     d1,loop           6
          rts                    12
*****
* check_address - searches through valid flash address ranges
*                to find which array is being accessed, and therefore
*                which set of control registers to use.
*
*                Note - flash register ranges are tested first, as they
*                have priority over an array that is mapped to the same
*                address.
*
* Entry - A0 contains address to be programmed
* Exit - A1 contains start address of register bank, or 0 if
*        no valid flash module found for address
*****
check_address  movem.l d0,-(a7)          push working reg for now
              move.l a0,d0             restrict address to 24 bits
              and.l  #00ffffff,d0
              move.l d0,a0

*
*          Is a0 within 16K register block?
ca_regs

          cmpa.l #FER_1&$ffffff,a0     range 1 start test..
          bcs   ca_2                   is a0 > range start?
          cmpa.l #(FER_1+FER_REGSZ-1)&$ffffff,a0
*                                     yes, now test against end..
          bhi   ca_2                   is a0 < range end?
          move.l #FER_1,a1             yes, within range
          bra   ca_good

*
*          Is a0 within 48K register block?
ca_2

          cmpa.l #FER_2&$ffffff,a0     range 2 start test..
          bcs   ca_3                   is a0 > range start?
          cmpa.l #(FER_2+FER_REGSZ-1)&$ffffff,a0
*                                     yes, now test against end..
          bhi   ca_3                   is a0 < range end?
          move.l #FER_2,a1             yes, within range
          bra   ca_good

*
*          Is a0 within 16K flash array?
ca_3

          move.l FEEBAH+FER_1,d0        read array1 start address
          and.l  #00ffffff,d0          clear d0[31:24]
          move.l d0,a1
          cmpa.l a1,a0
          bcs   ca_4                   is a0 > range start?
          add.l #FEE_SIZE_1-1,d0       calculate end addresses
          move.l d0,a1
          cmpa.l a1,a0
          bhi   ca_4                   is a0 < range end?
          move.l #FER_1,a1             yes, within range
          bra   ca_good

*
*          Is a0 within 48K flash array?
ca_4

          move.l FEEBAH+FER_2,d0        read array2 start address
          and.l  #00ffffff,d0          clear d0[31:24]
          move.l d0,a1
          cmpa.l a1,a0
          bcs   ca_bad                 is a0 > range start?
          add.l #FEE_SIZE_2-1,d0       calculate end addresses

```

```

        move.l  d0,a1
        cmpa.l  a1,a0
        bhi    ca_bad          is a0 < range end?
        move.l  #FER_2,a1      yes, within range
        bra    ca_good

*
*      No valid module being addressed - return 0 in A1
ca_bad
        movea.l #0,a1
ca_good
        movem.l (a7)+,d0
        rts

*****
* do_prog      - Programs one byte/word of data to memory
* Entry        - Target address in A0
*              byte or word data in D0
*              byte flag in d5 (non-zero => program byte data)
* Exit         - d0 contains difference between data to be programmed and read
*              back data ($00 if programming successful)
*              or $ff if address to be programmed is not recognised as flash
*              d3 is corrupted
*              a0 and d5 are unchanged
*****
do_prog
        bsr    dis_both        disable both modules (STOP)
        clr.w  d3              initialise pulse counter = 0
        bsr    check_address    get register address

        tst.l  a1              address OK?
        beq   dp_addrfail      no - bomb out

        move.w #flashen,FEEMCR(a1)  only enable module to be programmed
        move   #latch,FEECTL(a1)    enable verify + latch
        tst   d5                byte or word?
        beq   dp_word

*
*      ** Byte data to programming latch
        move.b d0,(a0)          write byte data to EEPROM
        bra   dp_prgloop

*
*      ** Word data to programming latch
dp_word
        move.w d0,(a0)          write word data to EEPROM

*
*      ** Initialise prog pulse time
dp_prgloop
        move.w #pwpp,d1        pulse time ready for usdelay

*
*      ** Programming stage
        move.w #prgen,FEECTL(a1)  enable prog voltage : set ENPE
        bsr.w  usdelay          wait pwpp microseconds

*
*      ** 'Off' time
        move.w #latch,FEECTL(a1)  disable voltage : clear ENPE
        move.w #tpr,d1           delay tpr microseconds after turning off vprog
        bsr   usdelay
        addq.w #1,d3            increment pulse count

*
*      ** Verify stage - store diff in d0
        moveq.l #0,d0           d0 ready to hold byte/word diff.
        tst   d5                byte or word?
        beq   dp_verw
        move.b (a0),d0          byte verify
        bra   dp_vertst
dp_verw
        move.w (a0),d0          word verify
dp_vertst
        beq   dp_margin        verify O.K?

*
*      ** Failed to verify
        cmpi.w #npp,d3         over max number of program pulses?
        bcs   dp_prgloop      no - continue

*
*      ** Failed to verify and max program time used
        move.w #norm,FEECTL(a1)  normal flash reads/writes
        bra   dp_end           return programming data error to caller

*
*      ** programmed OK - now re-program for the same number of pulses (100% margin)

```

```

dp_margin      subq.w #1,d3          compensate for extra dbcc loop
dp_mrgloop     move.w #pwpp,d1       set program pulse time
               move.w #prgen,FEECTL(a1) enable prog voltage : set ENPE
               bsr      usdelay      and delay
               move.w #latch,FEECTL(a1) disable voltage : clear ENPE
               move.w #tpr,d1       set program recovery time
               bsr      usdelay      and delay
               dbf      d3,dp_mrgloop count down pulses

*
** Check still programmed - store diff in d0
moveq.l #0,d0   d0 ready to hold byte/word diff.
tst    d5      byte or word?
beq    dp_verw2
move.b (a0),d0 byte verify
bra    dp_vertst2
dp_verw2      move.w (a0),d0       word verify
dp_vertst2    move.w #norm,FEECTL(a1) normal flash reads/writes
               bra    dp_end      return programmed data to caller
               (don't need to test)

*
** check_address address fail
dp_addrfail   move.w #$ff,d0       force fail because of bad address

*
** Fail + pass termination
dp_end        bsr      dis_both    disable both modules
               rts      and quit

*****
* Initialize - initialize routine is called by BD32 before any programming
* initialize and check main registers
* initialize global variables
* returns non-zero in D0 if can't continue with programming
*
* Exit - d0 cleared
*****
Initialize
* (Initialise modules but leave STOPped)

*
***** Initialisation module 1 main registers

move.w #flashdis,FEEMCR+FER_1 STOP module 1
move.w #$4,FEECTL+FER_1      make sure verify mode off

*
***** Initialisation module 2 main registers

move.w #flashdis,FEEMCR+FER_2 STOP module 2
move.w #$4,FEECTL+FER_2      make sure verify mode off

*
***** Now initialize globals

clr.l d0                      no error function return value
move #ProgOK,Error(A5)        initialise successful return value
rts                           done - return no error

*****
* ProgRecord - programs data from S-record buffer into EEPROM
* loops through the record, retrieving each byte/word and
* programming it at the specified S-record address + OffsetAddr
*
* IF ModeAddr == $1, OffsetAddr is calculated so that :
* (OffsetAddr + S-record address) = StartAddress
* (where StartAddress is user specified) and ModeAddr is then
* cleared
*
* Entry - no parameters: assumes S Record is in 'buffer'
* Exit - d0 is difference between data and EEPROM location
* (this will be 0 if programmed successfully)
* a0 will contain address at which program failed
* d5 will be non-zero if byte program, 0 if word program
*****
ProgRecord     movem.l a1/a2/d6,-(a7) save working registers
               lea.l  buffer(A5),a2  point to S-record buffer
               clr.l  d6
               move.b (a2)+,d6      get record type

               beq    prog_good      record type 0 (header)

```

```

*                               - exit as no data to program
      cmpi.b  #7,d6
      bcs    prog_start         record type 1,2 or 3 (code/data)
*                               - start programming

      bra    prog_good         record type >3, (not code/data)
*                               - exit as no data to program

prog_start  move.b  (a2)+,d6     get byte count from s-record
      subi.b  #4,d6           remove byte count due to address
      move.l  (a2)+,a0         get address (note : BD32 always
*                               stores 4 byte address field)

prog_offs   cmpi.w  #1,ModeAddr(A5)  Should we calculate offset?
      bne    prog_addoff
      move.l  a0,d5           put address in d5
      move.l  StartAddr(a5),a1     Yes, get desired start
      suba.l  d5,a1           ..use to calculate offset
      move.l  a1,OffsetAddr(a5)    ..store
      clr.w   ModeAddr(A5)      ..clear mode to signal done
prog_addoff adda.l  OffsetAddr(a5),a0  add offset to address

prog_1      move.l  a0,d5           store address in d5

      andi.l  #1,d5           mask all but bit 0
      bne    prog_2           program byte if odd address
      cmpi   #1,d6           count == 1?
      bne    prog_3           word program if not

* program byte data if address is odd or byte count is 1

prog_2      moveq   #1,d5         flag byte write
      move.b  (a2),d0         byte - get data
      bsr    do_prog         program byte/word
      tst.w  d0             programmed O.K?
      beq    prog_25

quit1       bsr    not_progd     no - does user want to quit?
      bne    prog_done

*
*   Either programmed O.K. (byte), or user wishes to continue
prog_25     addq.l  #1,a0         increment target address
      addq.l  #1,a2         increment buffer address
      subq   #1,d6         dec byte count
      bne    prog_1         loop till byte count = 0
      bra    prog_good       otherwise done

* program word data if address is even and byte count not equal to 1

prog_3      move.b  (a2)+,d0     get word - we don't know if
      asl.w  #8,d0           ..data in buffer is word aligned
      move.b  (a2)+,d0         ..so read two bytes

      bsr    do_prog         program byte/word
      tst.w  d0             programmed O.K?
      beq    prog_35

quit2       bsr    not_progd     no - does user want to quit?
      bne    prog_done

*
*   Either programmed O.K. (word), or user wishes to continue
prog_35     addq.l  #2,a0         increment target address
      subq   #2,d6         dec byte count
      bne    prog_1         loop till byte count = 0

prog_good   moveq.l #0,d0         no error
prog_done   movem.l (a7)+,a1/a2/d6  restore registers
      rts                    done

```

```

*****
* not_progd - informs user of programming error
* not_blank - informs user of blank check error
*          - user enters escape to stop, any other key to continue programming
* exit     - d0 is $0 and Z flag is set if user wants to continue
*          - d0 is non-zero, and Z flag is clear if user wants to abort
*****
not_progd:    bsr      Print
              dc.b    'prog: program fail at address $',0
              bra     n_bl

not_blank:    bsr      Print
              dc.b    'prog: EEPROM not blank, address $',0
n_bl         move.l   a0,d0          print address
              bsr      ltoh

              bsr      Print
              dc.b    13,10,'prog: Press <esc> to stop, any other to continue: ',7,0
              bsr      getchar
              move    d0,-(a7)      save char
              bsr      crlf
              move    (a7)+,d0      get char
              andi   #$ff,d0
              cmpi   #$1b,d0      escape?
              seq    d0            make d0 nonzero if so
              tst    d0            set SR for subsequent test
              rts

*****
* not_prog   - informs user of programming error
* Entry     - a0 contains fault address
*****
not_prog:    bsr      Print
              dc.b    'prog: program failed before $',0
              move.l  a0,d0          print address
              bsr      ltoh
              bsr      crlf
              rts

*****
* dis_both  - disables both flash EEPROM modules
* exit     - no registers modified
*****
dis_both     move.w   #flashdis,FEEMCR+FER_1  disable module 1 (set STOP)
              move.w   #flashdis,FEEMCR+FER_2  disable module 2 (set STOP)
              rts
              end

*****
* Prog msg  - message file for programming driver
*****

prog <filename> [<start>]    program M68F333 flash EEPROM from file
prog: Usage error: prog <filename> [<start address>]
prog: Error opening input file
prog: Error evaluating <start> address parameter
prog:
prog: End of file reached before S7/S8/S9 record was read
prog: S9 record read - file closed normally
prog: Checksum error in S-Record input file
prog: Format error in S-Record input file; file is probably not S-Records
prog: Programming error - check Vfpe / EEPROM is blank
prog: Unhandled exception encountered
prog: Programming completed O.K.

```

BULK — Erasure Driver

User Details

The BULK driver performs bulk erasure of a single flash EPROM module. The syntax used is:

```
BULK <module id>
```

The argument <module id> is used to specify the module to be erased. The value can be either '16' or '48' to specify the 16 kbyte or 48 kbyte Flash EEPROM modules respectively. A series of erasure passes are used. Each successive pulse is of progressively longer duration, until erasure is verified. Each erasure pass is indicated by the printing of a period, and if erasure is not verified after the maximum erasure time has been used, a bulk fail message is printed, along with the address of the first failed location.

As with the PROG driver, the BULK driver does not map the flash array to a particular address. The user must make certain that the array address does not conflict with addresses of other MCU modules, causing erasure to fail. The array can be relocated either by programming the shadow registers and then resetting the device, or by directly reconfiguring the base address registers. The base address registers can only be changed when the FLASH module LOCK bit is cleared.

Software Details

The BULK software applies erase pulses of increasing duration until the array and shadow registers verify as erased, then a final erase pulse is applied as an erase margin.

The source files for the BULK driver software are:

BULK.S62	Erase code source file
BULK.MSG	Message text file used by BD32
IPD.INC	Definitions required for the BD32 system calls

M68F333.INC MC68F333 constants definition file, including register addresses, other flash module information, and programming/erasure timing data. Timing information is compatible with the definitions used in the MC68F333 device specification to simplify updates.

Common include files used by both drivers are shown after the erasure driver code.

BULK Driver Listing

```

*****
* 'BULK' Resident Command Driver for MC68F333 device
*
* Utility to bulk erase an MC68F333 flash EPROM module
*
* Source file   : bulk.s62
* Object file  : bulk.d32
* Include files : M68F333.inc      (M68F333 addresses and programming constants)
*               ipd.inc          (BD32 system call constants)
* Message file  : bulk.msg
*
* Object file format: Motorola S-records
*
* Execute from BD32 as: bulk <module ID>
*   Module ID can be '16' or '48' and specifies which MC68F333 flash
*   module is to be bulk erased.
*
* Addressing modes : This code is designed as a driver for the BD32 background
*   debugger for CPU32 devices. A requirement is that the code must be
*   fully relocateable. All addresses (apart from fixed module addresses)
*   are relative, and where word alignment is not guaranteed, byte
*   accesses must be used.
*   Supervisor program space accesses are used when reading the flash
*   array to allow operation regardless of the configuration of the
*   flash modules's ASPC bits (FEEMCR).
*
* Word alignment : The embedded text strings have been adjusted in size so
*   that the following code remains word aligned - any modifications
*   to these strings should be adjusted accordingly. An assembler
*   'even' type directive to force word-alignment could be used if
*   available.
*****
*
*   Include files
*   lib      ipd.inc          BD32 call code definitions
*   lib      M68F333.inc      M68F333 device constants
*
*   BD32 return error codes : see file BULK.MSG for associated text

UsageError    equ    1          Usage: ...
BulkError     equ    2          error programming data
ExcepError    equ    3          unhandled exception
PassError     equ    4          erase successful

*
*   BD32 call return codes : see bd32 file BD32.DOC
SRecS9        equ    2          S9 Record - end of file

*
*   General constants
ErasedValue   equ    $ffff      erased state of EEPROM
sup_prog      equ    $6         supervisor/program space code

*
*   Flash control register constants
*   FEEMCR
flashdis      equ    $90c0      Module DISABLED, disable VFPE in BDM,
*                               no boot, unrestricted space, 2 cycle access
flashen       equ    $10c0      Module ENABLED, disable VFPE in BDM,
*                               no boot, unrestricted space, 2 cycle access
*   FEECTL
erase_on      equ    $7         Erase, VFPE enabled      (VFPE,ERAS,LAT,ENPE set)
erase_off     equ    $6         Erase, VFPE disabled   (VFPE,ERAS,LAT set)
norm          equ    $0         No programming/erase   (All cleared)

*
*   Variable area
section .data
dc.l Bulk          start address (add load offset)
ds.l 30            Stack area
*
*   initial stack pointer
stack

ModSize       dc.l $0          Module size
ModAddress    dc.l $0          Module address
StartAddress  dc.l $0          Start array address

Error         ds.w 1          error code

```

```

Era_shadow    dc.w    $9B00,$0000,$FFFF,$E000  erased shadow register mask
              dc.w    $0000,$0000,$0000,$0000  used for verification of erase
              dc.w    $FFFF,$FFFF,$FFFF,$FFFF
              dc.w    $0000,$0000,$0000,$0000

```

```

*****
*          CUSTOM VECTOR TABLE
*****
vectable     ds.l    13                      Alternate vector table

```

```

*****
*          EXCEPTION HANDLER ROUTINE
*          Use - Quits to BD32 with unhandled exception error code
*          Exception handling is included because many user errors
*          (mapping of flash/drivers etc) could cause bus errors,
*          f-line exceptions etc. Flash programming voltage is disabled
*          in case exception occurred during a programming cycle
*****

```

```

excep_h      move.w  #norm,FEECTL(a1)        normal flash reads/writes
*                                                    disable programming voltage
              move    #ExcepError,Error(A5)  unhandled excep error
              bra     Bulk_end

```

```

FileMode     dc.b    'r',0                  read mode for file open

```

```

*****
*          Execution start of driver 'BULK'
*          Entry (from BD32) :
*          d0 - number of driver parameters
*          a0 - address of parameter array
*          a5 - driver offset address
*****

```

```

Bulk
*          ***** Exception handler initialisation
              lea.l   vectable(PC),a1        get start of vector table
              movea.l a1,a2                  working (loop) copy
              lea.l   excep_h(PC),a3        get address of handler
              move.w  #$0c,d1                initialise copy loop
vecloop      move.l   a3,(a2)+              build new vector table
              dbf     d1,vecloop
              movec.l a1,vbr                set up vbr for new table

*          ***** SP and general register initialisation
              lea.l   stack(A5),a7          set up stack
              lea.l   stack(PC),a7          set up stack
              move.l  a0,a2                  get argv into a2
              move.l  d0,d2                  get argc into d2

*          ***** Print signon and warning message
              bsr     Print                  print signon message
              dc.b    'M68F333 Flash EEPROM Bulk Eraser Version 2.0 ',13,10,0

*          ***** Check command line
              cmpi   #2,d2                  argc = 2?
              beq    Bulk_1
              move   #UsageError,Error(A5)  arg count is wrong
              bra     Bulk_end

*          ***** Get module parameter, and use to set up ModAddress
Bulk_1
              move.l  #FER_1,ModAddress(a5)  assume 16K module initially
              move.l  #FEE_SIZE_1,ModSize(a5)
              addq.l  #4,a2                  skip over program name
              move.l  (a2)+,a0                get address of parameter

              move.b  (a0)+,d0                get two bytes of parameter
              asl.w   #8,d0                  (data in buffer may not be
              move.b  (a0)+,d0                word aligned so read 2 bytes)

```

```

        cmpi.w  #'16',d0          16k array specified?
        beq     Bulk11           yes, so O.K. to continue..
        move.l  #FER_2,ModAddress(a5) no, so first assume 48k
        move.l  #FEE_SIZE_2,ModSize(a5)
        cmpi.w  #'48',d0          ..and then verify
        beq     Bulk11           yes, so O.K. to continue..
        move    #UsageError,Error(A5) no, so flag useage error
        bra     Bulk_end         ..and quit

*
Bulk11      ***** Initialise module, and calculate array addresses
        bsr     Initialize       init hardware

*
          ***** Erase module now
        bsr     Erase
        tst.b   d0               was erase succesful?
        beq     Bulk_end

        move    #BulkError,Error(A5) no, so flag erase error
        bra     Bulk_end

*
Bulk_end    ***** Report any errors, exit back to BD32
        move    Error(A5),d1     get error code
        moveq.l #BD_QUIT,d0      exit program
        bgnd

*****
* FindStrEnd - searches an ASCII string for end of string
*             marker ('null'/ 0 char)
* Entry      - string pointed to by A0
* Exit       - returns a0 pointing to end of string marker
*             all other registers preserved
*****
FindStrEnd  move.w  d0,-(a7)      push temp register
          moveq   #-1,d0         max loop count 1st time thru
FSE_1       tst.b   (a0)+        byte == 0?
          dbeq   d0,FSE_1       uses loop mode
          bne    FSE_1          loop till test true
          subq.l #1,a0          decrement address reg.
          move.w (a7)+,d0       restore register
          rts

*****
* ntoh      - prints hex value of register D0 least sig nibble to screen
* Entry     - D0 contains nibble value
* Exit      - all registers preserved
*****
ntoh        movem.l d0/d1,-(a7)
          move.b  d0,d1
          andi.w  #$f,d1
          addi.b  #'0',d1
          cmpi.b  #10+'0',d1
          bcs    nt_1
          addi.b  #'A'-'9'-1,d1
nt_1        moveq  #BD_PUTCHAR,d0
          bgnd
          movem.l (a7)+,d0/d1
          rts

*****
* btoh      - prints hex value of byte register D0 to screen
* Entry     - D0 contains byte value
* Exit      - all registers preserved
*****
btoh        ror.b  #4,d0
          bsr     ntoh
          ror.b  #4,d0
          bsr     ntoh
          rts

```

```

*****
* wtoh          - prints hex value of word register D0 to screen
* Entry        - D0 contains word value
* Exit         - all registers preserved
*****
wtoh           ror.w   #8,d0
               bsr     btoh
               ror.w   #8,d0
               bsr     btoh
               rts

*****
* ltoh          - prints hex value of long word register D0 to screen
* Entry        - D0 contains long word value
* Exit         - all registers preserved
*****
ltoh           swap    d0
               bsr     wtoh
               swap    d0
               bsr     wtoh
               rts

*****
* Print         - prints constant string in code and returns to
*               program at first even location after string
* Entry        - parameters indexed from stacked return PC
* Exit         - stacked return PC modified to give correct return
*               all registers preserved
*****
Print          movem.l  a0/d0,-(a7)          save registers
               move.l  8(a7),a0            get address of string
*               (= stacked return address)
               moveq.l  #BD_PUTS,d0        function call
               bgnd
               bsr     FindStrEnd          get end of ASCII string
               move.l  a0,d0              test for odd address
               addq.l  #1,d0              skip past end of string
               btst   #0,d0
               beq    Print_1
               addq.l  #1,d0              it's odd - return to next addr
Print_1        move.l  d0,8(a7)            update stacked return address
               movem.l  (a7)+,d0/a0        get back registers
               rts                        done

*****
* crlf         - prints carriage return, line feed combo
* Entry        - no parameters
* Exit         - all registers preserved
*****
crlf          bsr     Print                carriage return, line feed
               dc.b   13,10,0,0
*               even
               rts

*****
* getchar      - returns character typed by user
* Entry        - no parameters
* Exit         - d0 contains character typed
*****
getchar       moveq.l  #BD_GETCHAR,d0
               bgnd
               rts

*****
* msdelay     - programmable milliseconds delay
* Entry        - delay time in ms in d1
*               legal values are 1 ... 65535
* Exit         - d1 corrupted
*               Note - routine calibrated for 16.78MHz clock / 2 clock memory
*****
msdelay       move.l  d2,-(a7)            preserve d2
               subq.w  #1,d1              compensate for dbcc offset of 1
               move.w  #$826,d2           initialise inner loop count to
*               compensate for entry overhead

```

```

loop      tst      d1
loop2    tst      d2
         dbf      d2,loop2
         move.w   #$82d,d2          inner loop count
         dbf      d1,loop
         move.l   (a7)+,d2         restore d2
         rts

```

```

*****
* Erase      - bulk erase routine
*             performs erase algorithm until maximum allowed erase pulses
*             used, or array has verified as correctly erased
* Entry      - module defined by ModAddress
*             ModSize
* Exit       - D0 is non zero if erase unsuccessful
*             A0 contains first error address if erase unsuccessful,
*             otherwise A0 corrupted
*             All other registers unchanged
*****

```

```

Erase
         movem.l  d1-d3/a1,-(a7)      preserve registers

*
         ***** Initialise timing and address parameters
         clr.w    d2                   initialise pulse counter k = 0
         clr.w    d3                   initialise cumulative erase time = 0
*                                     (used as erase margin)
         move.l   ModAddress(a5),a1    get module address into a1

         move.w   #erase_off,FEECTL(a1) set VFPE/ERAS/LAT
         move.w   d0,(a1)              write data to EEPROM

*
         ***** Erase cycle
db_1     bsr      Print                print 'progress dots'
         dc.b     '.', $0
         addq.w   #1,d2                increment pulse counter, k

*
         ***** Calculate erase time
         move.w   #tei,d1              erase pulse time = tei
         mulu.w   d2,d1                * k (ms) = d1
         add.w    d1,d3                add to cumulative time, d3

*
         ***** Apply erase pulse
         move.w   #erase_on,FEECTL(a1) enable prog voltage : set ENPE
         bsr     msdelay                wait tei*k milliseconds
         move.w   #erase_off,FEECTL(a1) disable voltage : clear ENPE

*
         ***** Recovery 'Off' time
         move.w   #ter,d1              delay erase recovery time, ter
         bsr     msdelay

*
         ***** Blank test array
         bsr     check_array            array now blank?
         tst.b   d0
         bne     db_q                  miss register test if array non-blank

*
         ***** Blank test shadow registers
         bsr     check_regs             registers now blank?
         tst.b   d0
         beq     db_2

*
         ***** Array and/or registers not blank
db_q     cmpi.w   #nep,d2              used max pulses, k>=nep?
         bcs     db_1                  no - continue

*
         ***** Fail, so print error message and quit
         move.w   #norm,FEECTL(a1)     yes - flag and quit
         bsr     Print
         dc.b     13,10,'bulk: erase failed address $',0
         move.l   a0,d0                print address
         bsr     ltoh
         bsr     crlf
         move.w   #$1,d0               flag error in d0
         bra     db_end

```

```

*          ***** Erase verifies OK - now add erase margin
db_2      move.w  d3,d1          erase margin time (em)
*          = sum of erase pulses = d3
          move.w  #erase_on,FEECTL(a1)  enable prog voltage : set ENPE
          bsr     msdelay          delay em
          move.w  #erase_off,FEECTL(a1)  disable voltage : clear ENPE

          move.w  #ter,d1          delay erase recovery time, ter
          bsr     msdelay

          move.w  #norm,FEECTL(a1)     normal accesses
          clr.l   d0                clear d0 to signal success

db_end    movem.l (a7)+,d1-d3/a1      restore registers
          rts

*****
* Initialize - initialize routine is called by BD32 before bulk erasing
*             initialize main flash registers
*             initialize global variables
*             returns non-zero in D0 if can't continue with programming
*
* Entry      - flash module address (register block) in ModAddress(a5)
*
* Exit       - d0 cleared
*             all other CPU registers preserved
*             flash array address written to StartAddress(a5)
*****
Initialize  movem.l a3-a4,-(a7)        preserve registers

*           (Initialise modules but leave STOPped)

*           ***** Initialisation and STOP module 1
          move.w  #flashdis,FEEMCR+FER_1  STOP module 1
          move.w  #norm,FEECTL+FER_1      make sure verify mode off

*           ***** Initialisation and STOP module 2
          move.w  #flashdis,FEEMCR+FER_2  STOP module 1
          move.w  #norm,FEECTL+FER_2      make sure verify mode off

*           ***** Start-up module to be erased, and get array addresses
          move.l  ModAddress(a5),a3        get module address into a3
          move.w  #flashen,FEEMCR(a3)     clear STOP
          movea.l FEEDBAH(a3),a4           get array start address
          move.l  a4,StartAddress(a5)      and store

          move    #PassError,Error(A5)    initialise to successfull erase code

          movem.l (a7)+,a3-a4            restore registers
          rts                             done - return no error

*****
* check_array - checks EEPROM array contents all are ErasedValue
*
* Entry      - StartAddress, ModSize parameters initialised
*
* Exit       - if array checks as ErasedValue
*             d0 = 0
*             a0 corrupted
*             D1 corrupted
*             else
*             d0 = 1
*             a0 = error address
*             d1 = error data
*****
check_array  movem.l d2,-(a7)            preserve registers
          move.l  #sup_prog,d0           configure array accesses as
          movec  d0,sfc                  ..supervisor/program space
          move.l  StartAddress(a5),a0    array start in a0
          move.l  ModSize(a5),d1         array size in d1

```

```

        asr.l   #1,d1           calculate array size in words
        subq.l  #1,d1           set up for dbcc loop
        move.w  #ErasedValue,d0 get erased value of EEPROM

bc_1    moves.w (a0)+,d2       get array word from supervisor/program space
        cmp.w   d2,d0          test (== ErasedValue?)
        dbne   d1,bc_1        loop while equal, and not end of array

        beq    bc_2           loop exit because of error?

        move.w  d2,d1          yes, put error data in d1,
        move.b  #$01,d0        and flag error, array not blank
        bra    bc_3

bc_2    clr.l   d0            no, flag no error, array tests OK

bc_3    movem.l (a7)+,d2      restore registers
        rts

```

```

*****
* check_regs - routine to blank check flash shadow registers for a module
*             with register start address specified in a0
*
* Entry      - a0 should contain register start address
* Exit       - if verified blank d0 = 0
*             else d0 = 1
*             d1= fault data
*             and a0 = fault address
*****

```

```

check_regs
        movem.l d2-d3/a1-a2,-(a7)   preserve registers
        move.l  ModAddress(a5),a0    get module address into a0
        move.l  a0,a2               use a2 as general pointer

*
* **** Check shadow registers against erased values table
        move.w  #15,d1              number of word checks (loop cnt.)
        lea.l   (Era_shadow,a5),a1   table address in a1

cr_loop
        move.w  (a2)+,d2            get a shadow register value,
        move.w  d2,d3              store,
        and.w   (a1),d2            ignore un-implemented bits,
        cmp.w   (a1)+,d2          and check erased..
        bne    cr_bad              O.K?
        dbf    d1,cr_loop          yes, loop if not finished
        clr.l   d0                 finished - signal blank check OK
        bra    cr_end              and return

*
* **** Un-erased shadow register found - notify and abort
cr_bad
        suba.l  #2,a2              get correct fault address
        move.w  d3,d1              and fault data
        move.w  #1,d0              flag fault

cr_end
        move.l  a2,a0              return fault address (if any)
        movem.l (a7)+,d2-d3/a1-a2  restore registers
        rts
        end

```

```

*****
* Bulk msg - message file for bulk erase driver
*****

```

```

BULK <16/48>           Bulk erase Orion 16k/48k EEPROM modules
bulk: usage error: BULK <16/48>
bulk: bulk erase failed
bulk: unhandled exception encountered
bulk: module erased O.K.

```

Initialization Files Used By Program and Erase Drivers

```
*****
* 'M68F333.INC' Define M68F333 addresses and programming constants
*****

FER_1      equ      $FFFFFF800      register block address for array #1 - 16k bytes
FEE_SIZE_1 equ      $4000           size of array #1 - 16k bytes
FER_2      equ      $FFFFFF820      registers block address for array #2 - 48k bytes
FEE_SIZE_2 equ      $c000           size of array #2 - 48k bytes
FER_REGSZ  equ      $20             size of register block (both arrays)

* register offsets

FEEMCR     equ      0               mod config register
FEETST     equ      2               test register
FEEBAH     equ      4               base address reg - high word
FEEBAL     equ      6               base address reg - low word
FEECTL     equ      8               program control reg
FEEBS0     equ      $10             bootstrap info 0
FEEBS1     equ      $12             bootstrap info 1
FEEBS2     equ      $14             bootstrap info 2
FEEBS3     equ      $16             bootstrap info 3

* bit assignments

STOP       equ      $8000
FRZ        equ      $4000
BOOT       equ      $1000
LOCK       equ      $800
ASPC1      equ      $200
ASPC0      equ      $100
WAIT1      equ      $80
WAIT0      equ      $40

FSTE       equ      $80
GADR       equ      $40
HVT        equ      $20
BTST       equ      $10
STRE       equ      2
MWPFF      equ      1

VFPE       equ      8
ERAS       equ      4
LAT        equ      2
ENPE       equ      1

* Flash EEPROM timing constants

* Programming constants
pwpp       equ      &20             program pulse width (us)
tpr        equ      &10             program recovery time (us)
npp        equ      &50             number of program pulses

* Erase constants
tei        equ      100             erase pulse increment time (ms)
ter        equ      1               erase recovery time (ms)
nep        equ      5               maximum number of erase pulses

* end of M68F333.inc
```



```

*****
* ipd.inc - equates for BD32 systems calls
*****

BD_QUIT          equ    0      quit - return to BD32
BD_PUTS          equ    1      puts - put string to console
BD_PUTCHAR       equ    2      putchar - print character on console
BD_GETS          equ    3      gets - get string from user
BD_GETCHAR       equ    4      getchar - get single character from user
BD_GETSTAT       equ    5      getstat - return 1 if character waiting from user
BD_FOPEN         equ    6      fopen - open disk file with specified mode
BD_FCLOSE        equ    7      fclose - close disk file
BD_FREAD         equ    8      fread - read from disk file
BD_FWRITE        equ    9      fwrite - write to disk file
BD_FTELL         equ   10      ftell - report current pointer position
BD_FSEEK         equ   11      fseek - seek disk file to given position
BD_FGETS         equ   12      fgets - read string from file
BD_FPUTS         equ   13      fputs - write string to file
BD_EVAL          equ   14      eval - evaluate arithmetic expression
BD_FREADSREC     equ   15      read s-record

* end of ipd.inc

```

PROGRAMMING/ERASURE EXAMPLES

The following examples show various program and erase operations. In all of the examples, keyboard input from the user is shown as **bold text**.

Example 1 - Programming The FLASH Modules

This example shows operations required to program the both 16 kbyte and 48 kbyte flash modules from their erased state. Programming data for the shadow registers is in the file TEST1R.0, while programming data for the arrays is in the file ARRAY64.0.

First, initialize MCU memory resources to allow the driver software to execute. In this case, the file SRAM-HIGH.DO is used to configure the on-chip TPURAM and SRAM.

```
BD32->do sramhigh.do
```

The file SRAMHIGH.DO initializes the device, and terminates by checking that the required memory resources are responding correctly. This is done by writing the first few bytes of TPURAM and SRAM, and then reading them back. The flash register blocks are also displayed.

The macro file prints the following results.

```

BD32->* Finished, should have TPURAM $100000 - $100e00,
BD32->*                      SRAM   $100e00 - $100fff
BD32->*                      SSP     $100ffe
BD32->*                      Drivers load @ $100000
BD32->*
BD32->*
BD32->* Test read of TPURAM:
BD32->md $100000 $10
00100000 5450 5520 5241 4D20 6D65 6D6F 7279 2020   TPU RAM memory
BD32->*
BD32->* Test read of SRAM:
BD32->md $100e00 $10
00100E00 5352 414D 206D 656D 6F72 7920 2020 2020   SRAM memory
BD32->*
BD32->* Flash register area:
BD32->md $fff800 $40

```

```

00FFF800  9BC0 0000 00FF C000 0000 0000 0000 0000  .@....@.....
00FFF810  FFFF FFFF FFFF FFFF 0000 0000 0000 0000  .....
00FFF820  9BC0 0000 00FF 0000 0000 0000 0000 0000  .@.....
00FFF830  FFFF FFFF FFFF FFFF 0000 0000 0000 0000  .....

```

NOTE

Ensure that the VFPE supply is enabled before the programming command is entered.

The base address registers are programmed to ensure that the array is correctly mapped:

```

BD32->prog test1r.0
($100036).....
Download completed OK - 53 records read
M68F333 Flash EEPROM Programmer Version 2.0
prog: Programming completed O.K.

```

At this point, the shadow registers are programmed with appropriate values, but the MCU must be reset for these to take effect. The initialization file SRAMHIGH.DO resets the MCU as one of its operations. If either of the flash modules have been programmed with the boot option enabled, it is best to disable them by holding DATA[15:14] low during rest.

```
BD32->do sramhigh.do
```

The macro file terminates with the following information:

```

BD32->* Finished, should have TPURAM $100000 - $100e00,
BD32->*          SRAM $100e00 - $100fff
BD32->*          SSP $100ffe
BD32->*          Drivers load @ $100000
BD32->*
BD32->* Test read of TPURAM:
BD32->md $100000 $10
00100000  5450 5520 5241 4D20 6D65 6D6F 7279 2020  TPU RAM memory
BD32->*
BD32->* Test read of SRAM:
BD32->md $100e00 $10
00100E00  5352 414D 206D 656D 6F72 7920 2020 2020  SRAM memory
BD32->*
BD32->* Flash register area:
BD32->md $fff800 $40

```

```

00FFF800  8200 0000 0000 0000 0000 0000 0000 0000  .....
00FFF810  0010 FFFE 0000 1000 0000 0000 0000 0000  ...~.....
00FFF820  8200 0000 0001 0000 0000 0000 0000 0000  .....
00FFF830  0010 FFFE 0001 1000 0000 0000 0000 0000  ...~.....

```

The dump of the flash control register blocks shows that the arrays are now mapped to \$00000 (16 kbyte) and \$10000 (48 kbyte). These addresses are correct for the array data file ARRAY64.0, which contains a full 64 kbytes of test data covering both arrays. Remember, the VFPE supply must remain enabled for programming to take place. ARRAY64.0 takes around 35 seconds to program.

```

BD32->prog array64.0
($100038).....
Download completed OK - 53 records read
M68F333 Flash EEPROM Programmer Version 2.0
prog: Programming completed O.K.

```

Programming is successful. Disable the VFPE supply if no more operations are required.

Example 2: Erasing The FLASH Modules

As with the programming example, the MCU is initialized to allow execution of the driver software, in this case by using the macro file SRAMHIGH.DO.

```
BD32->do sramhigh.do
```

The macro file terminates with the following information.

```
BD32->* Finished, should have TPURAM $100000 - $100e00,
BD32->*                               SRAM  $100e00 - $100fff
BD32->*                               SSP    $100ffe
BD32->*                               Drivers load @ $100000
BD32->*
BD32->*
BD32->* Test read of TPURAM:
BD32->md $100000 $10
00100000 5450 5520 5241 4D20 6D65 6D6F 7279 2020   TPU RAM memory
BD32->*
BD32->* Test read of SRAM:
BD32->md $100e00 $10
00100E00 5352 414D 206D 656D 6F72 7920 2020 2020   SRAM memory
BD32->*
BD32->* Flash register area:
BD32->md $fff800 $40

00FFF800 8200 0000 0000 0000 0000 0000 0000 0000   .....
00FFF810 0010 FFFE 0000 1000 0000 0000 0000 0000   ...~.....
00FFF820 8200 0000 0001 0000 0000 0000 0000 0000   .....
00FFF830 0010 FFFE 0001 1000 0000 0000 0000 0000   ...~.....
```

NOTE

Ensure that the MCU V_{FPE} supply is enabled before the erase command is entered.

The erase driver is then executed.

```
BD32->bulk 16
($100030).....
Download completed OK - 35 records read
M68F333 Flash EEPROM Bulk Eraser Version 2.0
.bulk: module erased O.K.
```

The message indicates that the erase is successful. The number of periods on the last message line indicates the number of erase passes used. In this instance, there is only one.

An erase failure results in the following message, which indicates the first address to fail erase verification. As before, the number of periods on the last message line indicates the number of erase passes used. In this case, five passes (the maximum number) are made before a failure is reported.

```
BD32->bulk 16
($100030).....
Download completed OK - 35 records read
M68F333 Flash EEPROM Bulk Eraser Version 2.0
....bulk: erase failed address $00000002
bulk: bulk erase failed
```

To erase the 48 kbyte array, the following command is used.

```
BD32->bulk 48
($100030).....
Download completed OK - 35 records read
M68F333 Flash EEPROM Bulk Eraser Version 2.0
.bulk: module erased O.K.
```

The erase is successful, with one erase pulse required. Disable V_{FPE} if no more operations are required.

Example 3 - Attempting To Erase A Conflicting Array

When the 16 kbyte array is mapped to its default erased address of \$FFFFC000, portions of the array coincide with other MCU register blocks, such as the ADC control registers, which start at \$FFFF700. Since control registers generally take precedence in the memory map, erasure will fail as the erase driver attempts to verify that the array is blank.

```
BD32->bulk 16
($100030).....
Download completed OK - 35 records read
M68F333 Flash EEPROM Bulk Eraser Version 2.0
.bulk: unhandled exception encountered
```

The failure indicated is an unhandled exception, but the results of any attempt to erase a conflicting array are unpredictable, and the operation should be prevented by remapping the array. This can be done either by modifying the base address in the FEEBAH and FEEBAL registers (if the LOCK bit is cleared), or by programming the module shadow registers and resetting the device.

Erasing the 48 kbyte array at the default address will not normally cause these problems, as it is mapped from \$FFFF0000 to \$FFFFBFFF, avoiding other MCU register areas.

```
BD32->bulk 48
($100030).....
Download completed OK - 35 records read
M68F333 Flash EEPROM Bulk Eraser Version 2.0
.bulk: module erased O.K.
```

FINDING ERRORS

Following are descriptions of errors that commonly occur during programming or erasure of FLASH modules using the BD32 drivers. Typical error messages and fixes are given in each case.

1. Flash array mapped over the BD32 driver area.

Error symptoms – The driver may hang, or terminate with a line \$F or non documented error.

To verify, use the BD32 DRIVER command to determine the BD32 driver execution address, and examine the FEEBAH and FEEBAL registers of the module being programmed/erased. If the driver is within the array area, either relocate the array (Example 1) or the BD32 driver execution address (Examples 1 and 3)

2. Flash array mapped over the flash module register area, or other registers.

Error symptoms – BULK fails to verify blank after the maximum erase time has been used, and prints the fail address. This address corresponds to the first register within the array area. The array may be fully erased in this case, only the verify mechanism fails. PROG will print a program fail error for the first array address being programmed that corresponds with a module register. It will be impossible to program this location as the register takes priority.

To verify, examine the FEEBAH and FEEBAL registers of the module being programmed/erased and ensure that the module array does not conflict with any other registers.

To fix, remap array, either manually (Example 1) or by programming shadow base registers (Example 3).

3. Attempting to program unimplemented shadow bits.

Error symptoms – PROG prints a program fail error for the shadow register address. The register may have been programmed correctly, but verify always fails.

To fix, make sure that programming data for unimplemented shadow bits is set to zero.

4. No VFPE supplied

Error symptoms – A PROG program fail occurs at the first location to be programmed. BULK fails to verify blank after the maximum erasure time.

To fix, apply the correct VFPE supply

5. FLASH module not erased

Error symptoms – A PROG program fail occurs at the first location which has bits to remain erased at one, that are already programmed to zero.

To fix, program to all zeroes, bulk erase, and reprogram.

THE DEMO PROGRAM

DEMO executes from the MC68F333 16 kbyte flash EEPROM array from reset. It displays information on an RS232 terminal connected to the MCU SCI port via a level shifter. Apart from the level shifter only internal resources are used, with the FLASH, TPURAM, and SRAM supplying all of the required memory. ANSI control codes are used to allow cursor movement and screen clearing.

The software is split into the files, DEMOA and DEMOR. DEMOA contains the code to be programmed into the flash array. DEMOR contains programming data for flash shadow registers (flash array mapped to \$00000000, flash enabled at reset, if reset logic state of DATA15 pin allows) and supplies the CPU32 boot information (SP = \$10ffe, PC = \$001000). Example 3 shows how these files are used

DEMO Program Code Listing

```
*****
* 'DEMOA' demo boot program for the 16K flash array, to be used with the
* register file 'DEMOR'
* Source file: 'DEMOA.S62'
* Object file: 'DEMOA.0'
* Object file format: Motorola S-records
*****
*          Character equates for terminal output
ESC       equ   $1b          Escape
CR        equ   $0d          Carr. return
LF        equ   $0a          Line feed
CRGT      equ   $1c          Cursor right
CLFT      equ   $1d          ,, left
CUP       equ   $1e          ,, up
CDN       equ   $1f          ,, down

*****
*          Main code - initializes system, and displays start up
*          message
*          Memory map:
*          $000000 - $004000 : 16K flash array (internal)
*          $010000 - $010dff : 3.5K TPURAM          ,,
*          $010e00 - $010fff : 0.5K SRAM           ,,
*****
          section .text
          org $1000

start
          move.w #$0100,$fffb04          TRAMBAR Set TPURAM base address
          move.w #$0000,$fffb00          TRAMMCR Unrestricted space
          move.w #$0e00,$fffb46          SRAMBAL Set SRAM base address
          move.w #$0001,$fffb44          SRAMBAH
          move.w #$0000,$fffb40          SRAMMCR Unrestricted, not locked
          move.l #$010ffe,a7            Initialize stack pointer
```

```

        move.w #$42cf,$fffa00          ,, SMCR
        move.w #$7f08,$fffa04          ,, SYPCR
        move.w #$0006,$fffa20          ,, SYPCR
        move.w #$0000,$fffa4e          ,, PFPAR
        move.w #$0000,$fffa4A          ,, CSORBT
        move.w #$0000,$fffa4E          ,, CSORO
        move.w #$0000,$fffa76          ,, CSOR10
        bsr    sciinit

*      MAIN ROUTINE
loop   bsr    clrscrm                    clear screen
        bsr    home                      home cursor
        bsr    printstring                Print 1st frame
        dc.b   ' * * * * * * * * * * ',CR,LF
        dc.b   ' flash EEPROM boot demo ',CR,LF
        dc.b   '* * * * * * * * * * ',CR,LF,0
        bsr    home                      home cursor
        bsr    printstring                Print 2nd frame
        dc.b   '* * * * * * * * * * ',CR,LF
        dc.b   ' flash EEPROM boot demo ',CR,LF
        dc.b   ' * * * * * * * * * * ',CR,LF,0
        bsr    home                      home cursor
        bsr    printstring                Print 3rd frame
        dc.b   ' * * * * * * * * * * ',CR,LF
        dc.b   '* flash EEPROM boot demo * ',CR,LF
        dc.b   ' * * * * * * * * * * ',CR,LF,0
        bra    loop                      and loop..

***** **
*      PRINTCHAR - Output a single character to SCI serial port
*      Entry    - Character in D0
*      Registers - B15 of D0 cleared only
***** **
printchar btst  #$0,$fffc0c            Ready for transmit (TDRE of SCSR)?
        beq    printchar                loop if not..
        move.w d0,$fffc0e              Send data (to SCDR)
        rts

***** **
*      PRINTSTRING- Output a string of characters to serial port
*                  defined by routine 'printchar'
*      Entry      - Character string resides at return PC address
*                  ie. after 'bsr printstring' command
*                  charcter string is terminated by null ($00)
*      Exit       - Program returns to word location after string
*                  end, no registers modified
*      Registers  - Stack (return address) modified
***** **
printstring
        movem.l a0/d0,-(a7)            Preserve a0,d0
        move.l  ($8,a7),a0             get return PC (address of string)
        moveq.l #$0,d0                clear all of d0
psloop  move.b  (a0)+,d0               get a char to print
        beq    psnull                 finish if null
        bsr    printchar
        bra    psloop                 and loop
psnull
*      ensure return PC is word aligned
        move.l  a0,d0
        btst   #0,d0
        beq    psok                   Already word aligned, so continue
        addq.l #$1,d0                 not aligned, so adjust

```

```

psok
    move.l d0,($8,a7)           Update return PC
    movem.l (a7)+,a0/d0       Recover a0,d0
    rts


***** **
*      CLRSCRN    - Clear screen by sending clear screen escapet
*                  sequence
*      Registers  - A0,D0 modified
***** **
clrscrn  bsr    printstring      'Clear Screen' escape sequence
         dc.b  ESC,[' ','2','J',$0,$0  ESC [ 2 J
         rts

***** **
*      HOME      - Move cursor to home position
*                  'home' escape sequence
*      Registers  - A0,D0 modified
***** **
home     bsr    printstring      'Home' escape sequence
         dc.b  ESC,[' ','0','H',$0,$0  ESC [ 0 H
         rts

***** **
*      SCIINIT   - SCI initialisation
***** **
sciinit
    move.w  #$0001,$fffc00      Initialize QMCR
    move.w  #$000f,$fffc04      QILR
    move.w  #$00f0,$fffc14      QPDR
    move.w  #$0000,$fffc16      QPAR
    move.w  #$0037,$fffc08      SCCR0
    move.w  #$000c,$fffc0a      SCCR1
    rts
    end

***** **
* 'DEMOR' boot program for the 16K flash array, to be used with the
* array file 'DEMOA'
* Source file: 'DEMOR.S62'
* Object file: 'DEMOR.0'
* Object file format: Motorola S-records
***** **
*      16K flash module register bank
*                  org $FFF800
         dc.w  $0200          FEE1MCR : STOP = 0
*                                     BOOT = 0
*                                     LOCK = 0
*                                     ASPC = %10
*
*                  org $FFF804
         dc.w  $0000          FEE1BAH
         dc.w  $0000          FEE1BAL (Base addr = $0000)
*                                     (range $0000-$4000)
*
*                  org $FFF810
         dc.l  $0010fffe      FEE1BS0/1 (Reset SP and PC)
         dc.l  $00001000      FEE1BS2/3
*                                     end

```

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and  are registered trademarks of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

How to reach us:

USA/EUROPE/Locations Not Listed: Motorola Literature Distribution;
P.O. Box 20912; Phoenix, Arizona 85036. 1-800-441-2447 or 602-303-5454

MFAX: RMFAX0@email.sps.mot.com – TOUCHTONE 602-244-6609
INTERNET: <http://Design-NET.com>

JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC, 6F Seibu-Butsuryu-Center,
3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 03-81-3521-8315

ASIA/PACIFIC: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,
51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298



MOTOROLA

1ATX31973-1 PRINTED IN USA 7/96 IMPERIAL LITHO 24611 2,500 AMCU YGAKAA

AN1255/D

