Using The Callable Routines In D-Bug12

By Gordon Doughman, Field Applications Engineer, Software Specialist

1 Introduction

All microcontrollers require some type of operating environment for the development and debugging of user software. One of the least expensive environments that can be provided for the software developer is a monitor/debugger program that executes in the target environment. Such a debugger, while providing an inexpensive environment for the controlled execution of developer software, does have some limitations. Because the monitor/debugger program executes out of ROM in the target environment, target resources are required for its execution. For this reason, the monitor does not provide true target system emulation. A ROM monitor, however, does provide some significant advantages over other debug environments.

In most cases, software developers require a stable environment to test new algorithms or conduct performance benchmarks. A ROM monitor can provide access to many internal utility routines that would otherwise have to be written by the software developer. In addition, a ROM monitor can provide default exception (interrupt) handlers that do not have to be written by the developer. These default exception handlers can provide graceful recovery if the developer's software inadvertently enables peripheral interrupts without providing an exception handler.

This application note provides the details necessary to utilize the D-Bug12 user-callable utility functions. Additionally, it shows how to substitute user interrupt service routines for D-Bug12's default exception handlers.

Note: The utility functions described in this application note are available in D-Bug12 version 1.x.x (for the MC68HC812A4) and version 2.x.x (for the MC68HC912B32). The location and size of the pointer table is different for the two versions. For version 1.x.x, the pointer table is located beginning at $FE00 and is 128 bytes long. For version 2.x.x, the pointer table is located beginning at $F680 and is only 64 bytes long. Addresses given in parentheses apply to D-Bug12 version 2.x.x.

2 User-Accessible Utility Routines

D-Bug12 currently provides access to eighteen different utility routines through an array of function pointers (addresses) beginning at $FE00 ($F680). Placing the table at a fixed address, allows access to the individual functions to remain constant even though the actual address of the routines may move when changes are made to the monitor. The table is 128 (64) bytes long, extending to $FE7F ($F6FF), allowing access to a maximum of 64 individual utility routines.

Because D-Bug12 was written almost entirely in C, the utility routines are presented as C function definitions. However, this does not mean that the utility routines are usable only when programming in C. They may easily be accessed when programming in assembly language as well. **Table 1** summarizes the available utility routines. A complete description of each utility routine is provided later in this application note.

Table 1 Utility Routines Summary

Function	Description	Pointer Address
main()	Start of D-Bug12	$FE00 ($F680)
getchar()	Get a character from SCI0 or SCI1	$FE02 ($F682)
putchar()	Send a character out SCI0 or SCI1	$FE04 ($F684)
printf()	Formatted Output — Translates binary values to characters	$FE06 ($F686)
GetCmdLine()	Obtain a line of input from the user	$FE08 ($F688)
sscanhex()	Convert an ASCII hexadecimal string to a binary integer	$FE0A ($F68A)
isxdigit()	Checks for membership in the set [0...9, a...f, A...F]	$FE0C ($F68C)
toupper()	Converts lower case characters to upper case	$FE0E ($F68E)
isalpha()	Checks for membership in the set [a...z, A...Z]	$FE10 ($F690)
strlen()	Returns the length of a null terminated string	$FE12 ($F692)
strcpy()	Copies a null terminated string	$FE14 ($F694)
out2hex()	Displays 8-bit number as two ASCII hex characters	$FE16 ($F696)
out4hex()	Displays 16-bit number as four ASCII hex characters	$FE18 ($F698)
SetUserVector()	Set up user interrupt service routine	$FE1A ($F69A)
WriteEEByte()	Write a data byte to on-chip EEPROM	$FE1C ($F69C)
EraseEE()	Bulk erase on-chip EEPROM	$FE1E ($F69E)
ReadMem()	Read data from the M68HC12 memory map	$FE20 ($F6A0)
WriteMem()	Write data to the M68HC12 memory map	$FE22 ($F6A2)

3 User-Accessible Function Calling Conventions

All of the user-accessible routines are written in C. In general, parameters are passed to the user-callable functions on the stack. Parameters must be pushed onto the stack in the reverse order they are listed in the function declaration (right-to-left) *except* for the last parameter (the first parameter listed in the C function declaration). The last parameter is passed to the function in accumulator D. Functions having only a single parameter pass it in accumulator D. Note that `char` parameters must always be converted to an `int`. This means that even if a parameter is declared as a `char` it will occupy two bytes of stack space as a parameter. Note also that `char` parameters should occupy the low order byte (higher byte address) of a word pushed onto the stack or accumulator B if the parameter is passed in D.

Parameters pushed onto the stack before the function is called remain on the stack when the function returns. It is the responsibility of the *calling* routine to remove passed parameters from the stack.

All 8- and 16-bit function results are returned in accumulator D. `char` values returned in accumulator D are located in the 8-bit accumulator B. `Boolean` function results are zero values for false and non-zero values for true.

None of the CPU12 register contents, except the stack pointer, are preserved by the called functions. If any of the register values need to be preserved, they should be pushed onto the stack before any of the parameters and restored after deallocating the parameters.

4 Assembly Language Interface

Calling the functions from assembly language is a simple matter of pushing the parameters onto the stack in the proper order and loading the first or only function parameter into accumulator D. The function can then be called with a JSR instruction. The code following the JSR instruction should remove any parameters pushed onto the stack. If a single parameter was pushed onto the stack, a simple PULX or PULY instruction is one of the most efficient ways to remove the parameter from the stack. If two or more parameters are pushed onto the stack, the LEAS instruction is the most efficient way to remove the parameters. Any of the CPU12 registers that were saved on the stack before the function parameters should be restored with corresponding PUL instructions.

An example of calling the `WriteEEByte()` function is shown below.

```
WriteEEByte:    equ     $FE1C               ; $F69C for v2.x.x
;
                .
                .
                .
                ldab    #$55                ; write $55 to EEPROM.
                pshd                        ; place the data on the stack.
                ldd     EEAddress           ; EEaddress to write data.
                jsr     [WriteEEByte,pcr]   ; Call the routine.
                pulx                        ; remove the parameter from stack.
                beq     EEWError            ; zero return value means error.
                .
                .
                .
```

The one part of the above example that requires an explanation is the addressing mode used by the JSR instruction. This addressing mode is a form of indexed indirect addressing that uses the program counter as an index register. The PCR mnemonic used in place of an index register name stands for *Program Counter Relative* addressing. In reality, the CPU12 does not support PCR. Instead, the PCR mnemonic is used to instruct the assembler to calculate an offset to the address specified by the label `WriteEEByte`. The offset is calculated by subtracting the value of the PC at the address of the first object code byte of the next instruction (in this case, PULX) from the address supplied in the indexed offset field (`WriteEEByte`). When the JSR instruction is executed, the opposite occurs. The CPU12 adds the value of the PC at the first object code byte of the next instruction to the offset embedded in the instruction object code. The indirect addressing, indicated by the square brackets, specifies that the address calculated as the sum of the index register (in this case the PC) and the 16-bit offset contains a pointer to the destination of the JSR.

If the assembler being used does not support program counter relative indexed addressing, the following two-instruction sequence can be used:

```
                ldx     WriteEEByte     ; load the address of WriteEEByte().
                jsr     0,x             ; Call the routine.
```

Listing 1 contains assembly language source macros that allow the routines to be easily called from assembly language. The code was written for Motorola's MCUasm macro assembler, however, only slight modification should be required to use the macros with other assemblers. Conspicuously absent from Listing 1 is a macro that supports the `printf()` function. Because `printf()` accepts a variable number of arguments, it is not possible to construct a macro to easily handle this situation with the Freescale MCUasm macro syntax.

Parameters are passed to the macros in the order they are declared in the C functions, left to right. The macros take care of passing the parameters to the functions in the proper order. When passing a parameter to a macro that represents the address of a constant or variable, the parameter must be preceded by the number or pound character (#). This tells the assembler to use the immediate addressing mode to pass the address of the parameter rather than the contents of the address indicated by the parameter. Listing 2 shows an example using the `sscanhex` macro.

4.1 Calling the User Accessible Routines from C

Because of the differences that may exist in the way various C compilers pass parameters, return function results, and deallocate local variables and parameters, accessing D-Bug12 user-callable functions from C can be a bit more complicated than calling them from assembly language.

If the compiler being used for code development follows the same function-calling conventions as the compiler used to develop D-Bug12, a minimum effort is required. The header file shown in Listing 3 may be `#included` with any source file that references D-Bug12 functions. The `#defines` at the end of

the header file are incorporated to allow the use of the standard function library names within the program text. Using the standard function library names will help ensure portability of the program text. In addition, by using the C preprocessor to replace the standard function library names with names prefixed by “DB12”, a program can use other functions contained in a standard function library without creating duplicate function conflicts in the linker. Listing 4 shows how D-Bug12’s `GetCmdLine()` and `printf()` function are used in a simple program.

If the compiler being used for code development does not follow the D-Bug12 function calling convention, assembly language “glue code” will have to be written for each D-Bug12 user-accessible function. The amount and complexity of the assembly language “glue code” will depend upon how closely the compiler follows the D-Bug12 function calling convention.

If, for example, a compiler passes all of its parameters onto the stack rather than passing the function’s first parameter in accumulator D, the assembly language “glue code” would first have to pull that parameter from the stack into accumulator D. It would then have to execute a JSR instruction to call the D-Bug12 function. Listing 5 shows an example of calling the `WriteEEByte()` function for a compiler that allows M68HC12 assembly language to be inserted directly into the C source code. If a compiler does not support this feature, the “glue code” will have to be assembled into an object file and combined with the compiled C source code with the compiler’s linker.

4.2 User Interrupt Service Routines

As mentioned previously, one of the advantages of D-Bug12 is its ability to provide default exception (interrupt) handlers. These default exception handlers can provide graceful recovery if software inadvertently enables peripheral interrupts. However, most developers will need to provide their own peripheral interrupt service handlers as part of the application development. The D-Bug12 `SetUserVector()` function allows a software developer to substitute his own interrupt service routines for any D-Bug12 default exception handler.

D-Bug12 accesses user interrupt service routines through a RAM-based interrupt vector table that mirrors CPU12 interrupt vectors which are located in EPROM from $FC00–$FFFF. When an enabled hardware interrupt occurs, a small interrupt service dispatch routine located in the D-Bug12 EPROM checks the corresponding entry in the RAM interrupt vector table. If the entry contains a value other than $0000, it is used as the address of the user’s interrupt service routine. If the corresponding RAM interrupt vector table entry contains an address of $0000, CPU control is returned to the D-Bug12 monitor where an exception message and CPU register contents are displayed.

User interrupt service routines may consist of a number of CPU12 instructions but must end with the RTI (return from interrupt) instruction. However, the maximum frequency at which interrupts occur will be restricted to something slightly less than when the user’s code is run from EPROM because of the small amount of code D-Bug12 must execute to determine if a user interrupt service routine is to be called. Before returning from the user’s interrupt service routine, the source of the interrupt must be cleared by writing to the interrupting peripheral’s control registers. If the interrupt source is not cleared before returning from the user’s service routine, the CPU12 will re-execute the same interrupt routine immediately after returning. The processor will become “stuck” in the interrupt service routine.

Listing 6 shows an example of how to use D-Bug12’s `SetUserVector()` function to provide an interrupt service routine that services a timer interrupt.

5 Callable Routine Descriptions

The following paragraphs contain complete descriptions and usage notes for D-Bug12 user-callable routines. In addition, the amount of stack space required by each routine and the routine’s pointer address are also supplied.

5.1 void main(void);

```
Pointer Address:     $FE00 ($F680)
Stack Space:         None
```

The first field in the table contains a pointer to the D-Bug12 `main()` function. This entry is provided for two purposes. First, the reset vector does not point to `main()` but rather to code that is contained in the file `Startup.s`. This file contains assembly language code that is required to initialize various hardware modules of the MC68HC812A4 before proper execution of the monitor can occur. As the monitor code is changed, the address of `main()` will change. Because the user may replace the supplied startup routines with his own startup code, he will have to examine the supplied D-Bug12 startup object code to determine the address of `main()`.

Placing the address of the `main()` function at the first location in the user-callable routines table allows user-supplied startup code to easily begin execution of the monitor with the simple instruction:

```
jmp     [$fe00,pcr]                 ; address is $f680 for v2.x.x
```

In addition, the user may want to execute a program stored in EEPROM or other non-volatile memory before entering the monitor. Again, for the same reasons listed above, providing the address of the monitor's `main()` function at a fixed address allows the location of `main()` to change without having to change the user's code.

Note: When executing a user program from power-up or reset that is stored in the on-chip EEPROM, the user's program should enter D-Bug12 through the startup code at the label "DEBUG12" rather than through `main()`. The `main()` function does not perform any hardware initialization and does not clear D-Bug12 variable memory.

When calling the `main()` function from a user program that began execution from D-Bug12, the user's program should first load the CPU12 stack pointer (SP) with the value "STACKTOP", which may be obtained from the file `Startup.s`.

Note: Reentering D-Bug12 from a user program through the `main()` function reinitializes all D-Bug12 internal tables and variables. Any previously set breakpoints will be lost and any breakpoint SWI's will remain in the user's program.

5.2 int getchar(void);

```
Pointer Address:     $FE02 ($F682)
Stack Space:         2 bytes
```

The `getchar()` function provides the ability to retrieve a single character from the control terminal SCI. If a character is not available in the SCI's receive data register when the function is called, the `getchar()` will wait until one is received. Because the character is returned as an `int`, the 8-bit character is placed in accumulator B.

5.3 int putchar(int);

```
Pointer Address:     $FE04 ($F684)
Stack Space:         4 bytes
```

The `putchar()` function provides the ability to send a single character to the control terminal SCI. If the SCI's transmit data register is full when the function is called, `putchar()` will wait until the transmit data register is empty before sending the character. No buffering of characters is provided. `putchar()` returns the character that was sent. However, it does not detect any error conditions that may occur in the process and therefore will never return `EOF`. Because the character is returned as an `int`, the 8-bit character is placed in accumulator B.

5.4 int printf(char *format,...);

```
Pointer Address:     $FE06 ($F686)
Stack Space:         Minimum of 64 bytes, does not include parameter
                     stack space.
```

The `printf()` function is used to convert, format, and print its arguments as standard output under control of the format string pointed to by `format`. It returns the number of characters that were sent to standard output. The version of `printf()` included as part of the monitor supports the formatted printing of all data types *except* floating point numbers.

The format string can contain two basic types of objects: ASCII characters which are copied directly from the format string to the display device, and conversion specifications that cause succeeding `printf()` arguments to be converted, formatted, and sent to the display device. Each conversion specification begins with a percent sign (%) and ends with a single conversion character. Optional formatting characters may appear between the percent sign and the conversion character in the following order:

```
[-][<FieldWidth>][.][<Precision>][h | l]
```

These optional formatting characters are explained in **Table 2**.

Table 2 Optional Formatting Characters

Character	Description
– (minus sign)	Left justifies the converted argument.
FieldWidth	Integer number that specifies the minimum field width for the converted argument. The argument will be displayed in a field at least this wide. The displayed argument will be padded on the left or right if necessary.
. (period)	Separates the field width from the precision.
Precision	Integer number that specifies the maximum number of characters to display from a string or the minimum number of digits for an integer.
h	To have an integer displayed as a short.
l (letter ell)	To have an integer displayed as a long.

The `FieldWidth` or `Precision` field may contain an asterisk (*) character instead of a number. The asterisk will cause the value of the next argument in the argument list to be used instead.

Table 3, shown below, contains the conversion characters supported by the `printf()` function included in D-Bug12. If the conversion character(s) following the percent sign are not one of the formatting characters shown in **Table 2** or the conversion characters shown in **Table 3**, the behavior of the `printf()` function is undefined.

Table 3 printf() Conversion Characters

Character	Argument Type; Displayed As
d, i	int; signed decimal number
o	int; unsigned octal number (without a leading zero)
x	int; unsigned hexadecimal number using abcdef for 10...15
X	int; unsigned hexadecimal number using ABCDEF for 10...15
u	int; unsigned decimal number
c	int; single character
s	char *; display from the string until a '\0'
p	void *; pointer (implementation-dependent representation)
%	no argument is converted; print a %

For those unfamiliar with C or the `printf()` function, the following examples show the results produced by the `printf()` function for several different format strings.

5.4.1 Example 1

```
printf("Signed Decimal: %d Unsigned Decimal: %u/n", Num, Num);
```

Where `Num` has the value $FFFF

Displays the result:

```
Signed Decimal: -1  Unsigned Decimal: 65535
```

5.4.2 Example 2

```
printf("Hexadecimal: %H Hexadecimal: %4.4H/n", Num, Num);
```

Where `Num` has the value $FF

Displays the result:

```
Hexadecimal: FF Hexadecimal: 00FF
```

5.4.3 Example 3

```
printf("This is a %s/n", TestStr);
```

Where `TestStr` is a *pointer* to (address of) the first byte of a null (zero) terminated character array containing `"Test"`.

Displays the result:

```
This is a Test
```

5.5 int GetCmdLine(char *CmdLineStr, int CmdLineLen);

```
Pointer Address:     $FE08 ($F688)
Stack Space:         11 bytes
```

The `GetCmdLine()` function is used to obtain a line of input from the user. `GetCmdLine()` accepts input from the user a single character at a time by calling `getchar()`. As each character is received it is echoed back to the user terminal by calling `putchar()` and placed in the character array pointed to by `CmdLineStr`. A maximum of `CmdLineLen - 1` printable characters may be entered. Only printable ASCII characters are accepted as input with the exception of the ASCII backspace character ($08) and the ASCII carriage return character ($0D). All other non-printable ASCII characters are ignored by the function.

The ASCII backspace character ($08) is used by the `GetCmdLine()` function to delete the previously received character from the command line buffer. When `GetCmdLine()` receives the backspace character, it will echo the backspace to the terminal, print the ASCII space character, $20, and then send a second backspace character to the terminal. This action will cause the previous character to be erased from the screen of the terminal device. At the same time, the character is deleted from the command line buffer. If a backspace character is received when there are no characters in `CmdLineStr`, the backspace character is ignored.

The reception of an ASCII carriage return character ($0D) terminates the reception of characters from the user. The carriage return, however, is not placed in the command line buffer. Instead an ASCII NULL character ($00) is placed in the next available buffer location.

Before returning, all the entered characters are converted to upper case. `GetCmdLine()` always returns an error code of `noErr`.

5.6 char * sscanhex(char *HexStr, unsigned int *BinNum);

```
Pointer Address:     $FE0A ($F68A)
Stack Space:         6 bytes
```

The `sscanhex()` function is used to convert an ASCII hexadecimal string to a binary integer. The hexadecimal string pointed to by `HexStr` may contain any number of ASCII hexadecimal characters. However, the converted value must be no greater than $FFFF. The string must be terminated by either an ASCII space ($20) or an ASCII NULL ($00) character.

The value returned by `sscanhex()` is either a pointer to the terminating character or a NULL pointer. A NULL pointer indicates that either an invalid hexadecimal character was found in the string or that the converted value of the ASCII hexadecimal string was greater than $FFFF.

5.7 int isxdigit(int c);

```
Pointer Address:     $FE0C ($F68C)
Stack Space:         4 bytes
```

The `isxdigit()`function tests the character passed in `c` for membership in the character set [0...9, a...f, A...F]. If the character `c` is part of this set, the function returns a non-zero (true) value; otherwise, a value of zero is returned.

5.8 int toupper(int c);

```
Pointer Address:     $FE0E ($F68E)
Stack Space:         4 bytes
```

If `c` is a lower-case character, [a...z], `toupper()` will return the corresponding upper-case letter. If the character is upper-case, it simply returns `c`.

5.9 int isalpha(int c);

```
Pointer Address:     $FE10 ($F690)
Stack Space:         4 bytes
```

The `isalpha()`function tests the character passed in `c` for membership in the character set [a...z, A...Z]. If the character `c` is part of this set, the function returns a non-zero (true) value; otherwise, a value of zero is returned.

5.10 unsigned int strlen(const char *cs);

```
Pointer Address:     $FE12 ($F692)
Stack Space:         4 bytes
```

The `strlen()` function returns the length of the string pointed to by `cs`. A string is an array of characters that is terminated by a `'\0'` character.

5.11 char * strcpy(char *s1, char *s2);

```
Pointer Address:     $FE14 ($F694)
Stack Space:         8 bytes
```

The `strcpy()` function copies the contents of string `s2` into the string pointed to by `s1` including the `'\0'`. A pointer to `s1` is returned.

5.12 void out2hex(unsigned int num);

```
Pointer Address:     $FE16 ($F696)
Stack Space:         70 bytes
```

The `out2hex()` function displays the lower byte of `num` on the control terminal as two hexadecimal characters. The upper byte of `num` is ignored. This function is provided for those that may not know how to use the `printf()` function. `out2hex()` simply calls `printf()` with a format string of "`%2.2X`".

5.13 void out4hex(unsigned int num);

```
Pointer Address:     $FE18 ($F698)
Stack Space:         70 bytes
```

`out4hex()` displays num on the control terminal as four hexadecimal characters. This function is provided for those that may not know how to use the `printf()` function. `out4hex()` simply calls `printf()` with a format string of "`%4.4X`".

5.14 int SetUserVector(int VectNum, Address UserAddress);

```
Pointer Address:     $FE1A ($F69A)
Stack Space:         8 bytes
```

The function `SetUserVector()` allows the user to substitute his own interrupt service routines for the default interrupt service routines provided by D-Bug12. Providing access to the RAM interrupt vector table *only* through this routine provides flexibility for future implementations of interrupt handling in D-Bug12. In addition, the memory location of the table may be changed without having to change user code. The address of the user's interrupt service routine, passed in `UserAddress`, should point to a routine that ends with an M68HC12 RTI instruction.

The following `enum typedef` defines the valid constants for `VectNum`. If an invalid constant is passed in `VectNum`, a value of –1 will be returned by `SetUserVector()`; otherwise a value of zero is returned.

Note: In early mask sets of the MC68HC812A4, the timer interrupts were incorrectly wired to the interrupt logic block. This caused the timer interrupt vectors to appear in the memory map at an incorrect address. To accommodate the changes made to fix the incorrect wiring of the timer interrupt logic, the constants passed to the `SetUserVector()` function in the `VectNum` parameter were changed. The constants shown below are to be used with D-Bug12 v1.0.4 and later and v2.0.0 and later. The constants contained within the comments should be used with D-Bug12 v1.0.2.

```
typedef Address char *;

typedef Byte unsigned char;

typedef enum Vect { UserPortHKWU = 7,
                    UserPortJKWU = 8,
                    UserAtoD = 9,
                    UserSCI1 = 10,
                    UserSCI0 = 11,
                    UserSPI0 = 12,
                    UserTimerCh0 = 23,      /* UserTimerCh0 = 13 */
                    UserTimerCh1 = 22,      /* UserTimerCh1 = 14 */
                    UserTimerCh2 = 21,      /* UserTimerCh2 = 15 */
                    UserTimerCh3 = 20,      /* UserTimerCh3 = 16 */
                    UserTimerCh4 = 19,      /* UserTimerCh4 = 17 */
                    UserTimerCh5 = 18,      /* UserTimerCh5 = 18 */
                    UserTimerCh6 = 17,      /* UserTimerCh6 = 19 */
                    UserTimerCh7 = 16,      /* UserTimerCh7 = 20 */
                    UserPAccOvf = 14,       /* UserPAccOvf = 21 */
                    UserPAccEdge = 13,      /* UserPAccEdge = 22 */
                    UserTimerOvf = 15,      /* UserTimerOvf = 23 */
                    UserRTI = 24,
                    UserIRQ = 25,
                    UserXIRQ = 26,
                    UserSWI = 27,
                    UserTrap = 28,
                    RAMVectAddr = -1 };
```

Once set, all of the addresses of the user's interrupt service routines will remain in the RAM vector table until D-Bug12 is restarted by a hardware reset. Alternately, individual interrupt service routine addresses may be removed by passing a null pointer in the `UserAddress` parameter.

Passing the constant 'RAMVectAddr' in the VectNum parameter will return the base address of the RAM interrupt vector table instead of an error code. This will allow the user to make numerous changes to the RAM vector table without having to call the SetUserVector() function for each interrupt vector change. When accessing the RAM vector table by using the base address, the Vect enumerated constants must be multiplied by two before being used as an offset into the RAM vector table.

Note: Care should be used when allowing addresses of user interrupt service routines to remain in the RAM vector table. If the addresses of interrupt service routines change during program development, the D-Bug12 interrupt handler will most probably jump to an incorrect program address resulting in loss of CPU/monitor control.

5.15 Boolean WriteEEByte (Address EEAddress, Byte EEData);

```
Pointer Address:     $FE1C ($F69C)
Stack Space:         12 bytes
```

The WriteEEByte() function provides a mechanism to program individual bytes of the on-chip EEPROM without having to manipulate the EEPROM programming control registers. WriteEEByte() does not perform any range checking on EEAddress to ensure that it falls within the address range of the on-chip EEPROM. A user program can determine the start address and size of the on-chip EEPROM array by examining the data contained in the custom data area fields CustData.EEBase and CustData.EESize.

A byte erase operation is performed before the programming operation and a verify is performed after the programming operation. If the EEPROM data does not match EEData, a false (zero value) is returned by the function.

5.16 int EraseEE(void);

```
Pointer Address:     $FE1E ($F69E)
Stack Space:         4 bytes
```

The EraseEE() function provides a mechanism to bulk erase the on-chip EEPROM without having to manipulate the EEPROM programming control registers. After the bulk erase operation is performed, the memory range described by CustData.EEBase and CustData.EESize is checked for erasure. If any of the bytes does not contain 0xff, a non-zero error code is returned.

5.17 int ReadMem (Address StartAddress, Byte *MemDataP, unsigned int NumBytes);

```
Pointer Address:     $FE20 ($F6A0)
Stack Space:         10 bytes
```

The ReadMem() function is used internally by D-Bug12 for all memory read accesses. For this implementation of the monitor, the ReadMem() function simply reads NumBytes of data directly from the target memory and places it in a buffer pointed to by MemDataP. A user-implemented command would probably not benefit from the use of this function. Instead, it could read values directly from memory. A non-zero error code is returned if a problem occurs while reading target memory.

5.18 int WriteMem (Address StartAddress, Byte *MemDataP, unsigned int NumBytes);

```
Pointer Address:     $FE22 ($F6A2)
Stack Space:         22 bytes
```

The WriteMem() function is used internally by D-Bug12 for all memory write accesses. WriteMem() is different from ReadMem() in that it is aware of the on-chip EEPROM memory. If a byte is written to the memory range described by CustData.EEBase and CustData.EESize, WriteMem() calls the WriteEEByte() function to program the data into the on-chip EEPROM memory. A non-zero error code is returned if a problem occurs while writing target memory.

6 Program Listings

6.1 Listing 1 — Assembly Language Source Macros Allowing Routines to be Easily Called from Assembly language

```
;
;             The label "Version" should be set to the D-Bug12 version number that is being used.
;             For example, if the D-Bug12 version number is 1.0.4, Version should be set to 104.
;
Version:      equ 104
;
              if  Version=102
;
;
;             constants used with the SetUserVector() function to set the address of user supplied
;             interrupt service routines. These constants are only for use with D-Bug12 version 1.0.2.
;
UserPortHKWU:equ 7        ; PortH key wake-up user interrupt
UserPortJKWU:equ 8        ; PortJ key wake-up user interrupt
UserAtoD:     equ 9       ; A-to-D user interrupt
UserSCI1:     equ 10      ; SCI #1 user interrupt (not available in the MC68HC912B32)
UserSCI0:     equ 11      ; SCI #0 user interrupt
UserSPI0:     equ 12      ; SPI #0 user interrupt
UserTimerCh0:equ 13       ; Timer Channel #0 user interrupt
UserTimerCh1:equ 14       ; Timer Channel #1 user interrupt
UserTimerCh2:equ 15       ; Timer Channel #2 user interrupt
UserTimerCh3:equ 16       ; Timer Channel #3 user interrupt
UserTimerCh4:equ 17       ; Timer Channel #4 user interrupt
UserTimerCh5:equ 18       ; Timer Channel #5 user interrupt
UserTimerCh6:equ 19       ; Timer Channel #6 user interrupt
UserTimerCh7:equ 20       ; Timer Channel #7 user interrupt
UserPAccOvf: equ 21       ; Pulse Accumulator Overflow user interrupt
UserPAccEdge:equ 22       ; Pulse Accumulator Edge user interrupt
UserTimerOvf:equ 23       ; Timer counter overflow user interrupt
UserRTI:      equ 24      ; Real Time Interrupt user interrupt
UserIRQ:      equ 25      ; CPU Maskable Interrupt request user interrupt
UserXIRQ:     equ 26      ; CPU Non-maskable Interrupt request user interrupt
UserSWI:      equ 27      ; Software Interrupt user interrupt
UserTrap:     equ 28      ; Instruction Trap user interrupt
RAMVectAddr: equ -1       ; returns the base address of the RAM interrupt vector table.
;
              else
;
;             constants used with the SetUserVector() function to set the address of user supplied
;             interrupt service routines. These constants are for use with D-Bug12 version 1.0.4 and later
;             or version 2.0.0 and later.
;
UserPortHKWU:equ 7        ; PortH key wake-up user interrupt
UserPortJKWU:equ 8        ; PortJ key wake-up user interrupt
UserAtoD:     equ 9       ; A-to-D user interrupt
UserSCI1:     equ 10      ; SCI #1 user interrupt (not available in the MC68HC912B32)
UserSCI0:     equ 11      ; SCI #0 user interrupt
UserSPI0:     equ 12      ; SPI #0 user interrupt
UserPAccEdge:equ 13       ; Pulse Accumulator Edge user interrupt
UserPAccOvf: equ 14       ; Pulse Accumulator Overflow user interrupt
UserTimerOvf:equ 15       ; Timer counter overflow user interrupt
UserTimerCh7:equ 16       ; Timer Channel #7 user interrupt
UserTimerCh6:equ 17       ; Timer Channel #6 user interrupt
UserTimerCh5:equ 18       ; Timer Channel #5 user interrupt
UserTimerCh4:equ 19       ; Timer Channel #4 user interrupt
UserTimerCh3:equ 20       ; Timer Channel #3 user interrupt
UserTimerCh2:equ 21       ; Timer Channel #2 user interrupt
UserTimerCh1:equ 22       ; Timer Channel #1 user interrupt
UserTimerCh0:equ 23       ; Timer Channel #0 user interrupt
UserRTI:      equ 24      ; Real Time Interrupt user interrupt
UserIRQ:      equ 25      ; CPU Maskable Interrupt request user interrupt
UserXIRQ:     equ 26      ; CPU Non-maskable Interrupt request user interrupt
UserSWI:      equ 27      ; Software Interrupt user interrupt
UserTrap:     equ 28      ; Instruction Trap user interrupt
RAMVectAddr: equ -1       ; returns the base address of the RAM interrupt vector table.
;
;
              endif
;
;
              if  Version<200; if we're assembling for version 1.x.x
TableBase:    equ $fe00    ; the address table is located at $fe00
              else
TableBase:    equ $f680    ; for version 2.x.x the table is located at $f680
              endif
;
;
jmain:        equ TableBase+$00
jgetchar:     equ TableBase+$02
jputchar:     equ TableBase+$04
printf:       equ TableBase+$06
jGetCmdLine: equ TableBase+$08
jsscanhex:    equ TableBase+$0a
jisxdigit:    equ TableBase+$0c
jtoupper:     equ TableBase+$0e
jisalpha:     equ TableBase+$10
jstrlen:      equ TableBase+$12
```

```
jstrcpy:     equ TableBase+$14
jout2hex:    equ TableBase+$16
jout4hex:    equ TableBase+$18
jSetUsrVect: equ TableBase+$1a
jWriteEEByte:equ TableBase+$1c
jEraseEE:    equ TableBase+$1e
jReadMem     equ TableBase+$20
jWriteMem:   equ TableBase+$22
;
;            C function: void main(void);
;
main:        macro
             jmp [jmain,pcr]      ; start D-Bug12 from main().
             endm

;
;
;
;            C function: int getchar(void);
;
getchar:     macro
             jsr [jgetchar,pcr]; call D-Bug12's getchar routine. return the character in the B-accumulator.
             endm

;
;
;
;            C function: int putchar(int);
;
putchar:     macro
             ldab\1               ; load the character to send into the B-accumulator.
             jsr [jputchar,pcr]; call D-Bug12's getchar routine. sent character is returned in B-accumulator.
             endm

;
;            C function: int GetCmdLine(char *CmdLineStr, int CmdLineLen);
;
GetCmdLine:  macro
             ldd \2               ; load the length of the command line character buffer.
             pshd                 ; place it on the stack.
             ldd \1               ; get a pointer to the character buffer
             jsr [jGetCmdLine,pcr]; go get characters from the user.
             pulx                 ; remove the command line length parameter from the stack.
             endm

;
;            C function: char * sscanhex(char *HexStr, unsigned int *BinNum);
;
sscanhex:    macro
             ldd \2               ; get a pointer to a word location where the conversion result will be placed
             pshd                 ; place it on the stack.
             ldd \1               ; get a pointer to the ASCII hex string to convert.
             jsr [jsscanhex,pcr]; go convert ASCII hex string to binary
             pulx                 ; 1-byte inst. to remove the pointer to the conversion result from the stack.
             endm

;
;            C function: int isxdigit(int c);
;
isxdigit:    macro
             ldab\1               ; load ASCII character into the B-accumulator.
             jsr [jisxdigit,pcr]; go check for membership in the character set 0..9, A..F, a..f.
             endm

;
;            C function: int toupper(int c);
;
toupper:     macro
             ldab\1               ; load ASCII character into the B-accumulator.
             jsr [jtoupper,pcr]; convert the character to upper case if the character is in the set a..z.
             endm

;
;            C function: int isalpha(int c);
;
isalpha:     macro
             ldab\1               ; load ASCII character into the B-accumulator.
             jsr [jisalpha,pcr]; go check for membership in the character set A..Z, a..z.
             endm

;
;            C function: unsigned int strlen(const char *cs);
;
strlen:      macro
             ldd \1               ; get a pointer to the null ('\0') terminated character array.
             jsr [jstrlen,pcr]    ; go count the number of characters in the string.
             endm

;
;            C function: char * strcpy(char *s1, char *s2);
;
strcpy:      macro
             ldd \2               ; get pointer to source string (s2) onto the stack.
             pshd                 ; place it on the stack.
             ldd \1               ; get pointer to destination string (s1)
             jsr [jstrcpy,pcr]    ; go copy the string.
             pulx                 ; one byte instruction to remove the source string pointer (s2)
                                  ; from the stack.
             endm

;
;            C function: void out2hex(unsigned int num);
;
out2hex:     macro
             ldab\1               ; get the 8-bit byte to display as ASCII hex.
```

```
            jsr [jout2hex,pcr]; go display the byte.
            endm
;
;           C function: void out4hex(unsigned int num);
;
out4hex:    macro
            ldd \1              ; get the 16-bit word to display as ASCII hex.
            jsr [jout4hex,pcr]; go display the word.
            endm
;
;           C function: int SetUserVector (int VectNum, Address UserAddress);
;
SetUserVector:macro
            ldd \2              ; get the address of the users interrupt service routine.
            pshd                ; place it on the stack.
            ldd \1              ; get the interrupt vector to set.
            jsr [jSetUsrVect,pcr]; go set the user's interrupt vector.
            pulx                ; one byte instruction to remove the address of the users
                                ; interrupt service routine from the stack.
            endm
;
;           C function: Boolean WriteEEByte(Address EEAddress, Byte EEData);
;
WriteEEByte: macro
            ldab\2              ; get the data byte to place in EEPROM memory.
            pshd                ; place it on the stack.
            ldd \1              ; get the EEPROM byte address.
            jsr [jWriteEEByte,pcr]; go program the EEPROM byte.
            pulx                ; one byte instruction to remove the EEPROM data from the stack.
            endm
;
;           C function: int EraseEE(void);
;
EraseEE:    macro
            jsr [jEraseEE,pcr]; go bulk erase the EEPROM.
            endm
;
;           C function: int ReadMem (Address StartAddress, Byte *MemDataP, unsigned int NumBytes);
;
ReadMem:    macro
            ldd \3              ; get the number of bytes to read.
            pshd                ; place it on the stack.
            ldd \2              ; get a pointer to a buffer in which to place the memory contents.
            pshd                ; place it on the stack.
            ldd \1              ; get the memory address where we will start reading data.
            jsr [jReadMem,pcr]; go read data from the target memory.
            leas4,s             ; remove the 2 parameters placed on the stack.
            endm
;
;           C function: int WriteMem (Address StartAddress, Byte *MemDataP, unsigned int NumBytes);
;
WriteMem:   macro
            ldd \3              ; get the number of bytes to write.
            pshd                ; place it on the stack.
            ldd \2              ; get a pointer to the data that will be placed in memory.
            pshd                ; place it on the stack.
            ldd \1              ; get the memory address where we will start writing data to memory.
            jsr [jWriteMem,pcr]; go write our data into the target memory.
            leas4,s             ; remove the 2 parameters placed on the stack.
            endm
;
;
```

6.2 Listing 2 — Example Using the sscanhex Macro

```
;
;       This small program demonstrates the use of several of the user callable functions
;       contained in D-Bug12. The routine performs the following functions:
;
;       1.) Displays a prompt by calling printf()
;       2.) Accepts an ASCII hexadecimal number typed by the user by calling GetCmdLine()
;       3.) Converts the entered number to binary by calling sscanhex()
;       4.) Displays the entered hexadecimal number as Signed Decimal, Hexadecimal, and Unsigned decimal number.
;       5.) Returns to D-Bug12 when a blank line is entered.
;
;
        include  “DB12Macs.Asm”
        opt      lis
;
Space   equ      $20             ; space character.
;
        org      $800
Buffer  ds       20              ; character buffer for user input
BufferP ds       2               ; pointer into the character buffer.
BinNum  ds       2               ; converted hex number.
;
;
        org      $7000
;
Test:   ldd      #PromptStr      ; load a pointer to the prompt string.
        jsr      [printf,pcr]; go print the prompt.
        GetCmdLine#Buffer,#20    ; now, go get an ASCII hex number from the user.
        ldx      #Buffer         ; point to the start of the buffer.
SkipSpcs:ldaa    0,x             ; get a character from the buffer.
        cmpa     #Space          ; leading space character?
        bne      DoneSkip        ; no, we're done.
        inx                      ; yes. point to the next character in the buffer.
        bra      SkipSpcs        ; go check for another space character.
DoneSkip:stx     BufferP         ; save a pointer to the first non-blank character in the buffer.
        tst      0,x             ; check to see if a blank line was entered.
        beq      Done            ; return to D-Bug12 if done.
        sscanhex BufferP,#BinNum; convert the number from ascii to binary.
        cpd      #0
        bne      PrtResult
        ldd      #Error          ; load a pointer to the error string.
        jsr      [printf,pcr]; go print the error message.
        bra      Test
PrtResult:ldd    BinNum          ; get the value of the converted number.
        pshd                     ; place three copies of the number on the stack. one for “%u”
        pshd                     ; one for “%4.4X”
        pshd                     ; one for “%d”
        ldd      #ResultStr
        jsr      [printf,pcr]
        leas     6,s
        bra      Test
Done:   swi
;
;
PromptStr:db     $0d, $0a,”Enter a Hex number: “,0
Error:  db       $0d, $0a,”Invalid hexadecimal number entered.”,$0d, $0a,0
ResultStr:db     $0d, $0a, $0d, $0a,”Signed Decimal = %d”,$0d, $0a,”Hexadecimal = %4.4X”,$0d, $0a,”Unsigned dec-
imal = %u”,$0d, $0a,0
```

6.3 Listing 3 — Header File that May be #included with Any Source File that References D-Bug12 Functions

```
/* This file may be #included with any C source file that uses the D-Bug12 */
/* user callable routines. It provides a simple, portable way to access the */
/* routines from C without having to use any assembly language “glue code” */

/* some typedefs used by D-Bug12 */

typedef char * Address;
typedef int Boolean;
typedef unsigned char Byte;

/*
        The symbol “Version” should be set to the D-Bug12 version number that is being used.
        For example, if the D-Bug12 version number is 1.0.4, Version should be set to 104.
*/

#define Version 104

#if Version == 102

typedef enum Vect { UserPortHKWU = 7,
                    UserPortJKWU = 8,
                    UserAtoD = 9,
                    UserSCI1 = 10,
                    UserSCI0 = 11,
                    UserSPI0 = 12,
                    UserTimerCh0 = 13,
                    UserTimerCh1 = 14,
                    UserTimerCh2 = 15,
                    UserTimerCh3 = 16,
                    UserTimerCh4 = 17,
                    UserTimerCh5 = 18,
                    UserTimerCh6 = 19,
                    UserTimerCh7 = 20,
                    UserPAccOvf = 21,
                    UserPAccEdge = 22,
                    UserTimerOvf = 23,
                    UserRTI = 24,
                    UserIRQ = 25,
                    UserXIRQ = 26,
                    UserSWI = 27,
                    UserTrap = 28,
                    RAMVectAddr = -1 };

#else

typedef enum Vect { UserPortHKWU = 7,
                    UserPortJKWU = 8,
                    UserAtoD = 9,
                    UserSCI1 = 10,
                    UserSCI0 = 11,
                    UserSPI0 = 12,
                    UserPAccEdge = 13,
                    UserPAccOvf = 14,
                    UserTimerOvf = 15,
                    UserTimerCh7 = 16,
                    UserTimerCh6 = 17,
                    UserTimerCh5 = 18,
                    UserTimerCh4 = 19,
                    UserTimerCh3 = 20,
                    UserTimerCh2 = 21,
                    UserTimerCh1 = 22,
                    UserTimerCh0 = 23,
                    UserRTI = 24,
                    UserIRQ = 25,
                    UserXIRQ = 26,
                    UserSWI = 27,
                    UserTrap = 28,
                    RAMVectAddr = -1 };

#endif

/* structure that defines the functions in D-Bug12's user accessible */
/* function table. Also provides a function prototype for each function */
/* Documentation for each of these functions can be found in Application */
/* Note AN-xxxx text */

typedef struct {
                 void (*DB12main)(void);
                 int (*DB12getchar)(void);
                 int (*DB12putchar)(int);
```

```
                int (*DB12printf)(const char *,...);
                int (*GetCmdLine)(char *CmdLineStr, int CmdLineLen);
                char * (*sscanhex)(char *HexStr, unsigned int *BinNum);
                int (*DB12isxdigit)(int c);
                int (*DB12toupper)(int c);
                int (*DB12isalpha)(int c);
                unsigned int (*DB12strlen)(const char *cs);
                char * (*DB12strcpy)(char *s1, char *s2);
                void (*out2hex)(unsigned int num);
                void (*out4hex)(unsigned int num);
                int (*SetUserVector)(int VectNum, Address UserAddress);
                Boolean (*WriteEEByte)(Address EEAddress, Byte EEData);
                int (*EraseEE)(void);
                int (*ReadMem)(Address StartAddress, Byte *MemDataP, unsigned int NumBytes);
                int (*WriteMem)(Address StartAddress, Byte *MemDataP, unsigned int NumBytes);

            } UserFN, * UserFNP;

/* defines a pointer to the start of D-Bug12's user accessible functable */

#if Version < 200

#define MyUserFNP ((UserFNP)0xfe00)/* in D-Bug12 version 1.x.x the user accessible table begins at $fe00 */

#else

#define MyUserFNP ((UserFNP)0xf680)/* in D-Bug12 version 2.x.x the user accessible table begins at $f680 */

#endif

/* The following #defines are used to provide for portability and avoid a linker */
/* conflict with the standard library functions of the same name. No #define is */
/* included for DB12main() since all C programs must contain a main() function */

#define printf DB12printf
#define getchar DB12getchar
#define putchar DB12putchar
#define isxdigit DB12isxdigit
#define toupper DB12toupper
#define isalpha DB12isalpha
#define strlen DB12strlen
#define strcpy DB12strcpy
```

6.4 Listing 4 — Example of `GetCmdLine()` and `printf()` Functions

```
#include “DBug12.h”

/*****************************************************************************/

void main(void)

 {

 /* Variable Declarations */

 char CmdLine[40];               /* used to hold the command line string */

 /* Begin Function main() */

   do
    {
     DBug12FNP->printf(“\n\r>”);/* display a prompt */

     DBug12FNP->GetCmdLine(CmdLine, 40);/* get a line of input from the user */

     DBug12FNP->printf(“\n\r”);/* go to the next line on the screen */

     DBug12FNP->printf(CmdLine);/* echo the line back to the user */
  }
   while (*CmdLine!= 0)   /* continue until a blank line is entered */

}       /* end main */

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

6.5 Listing 5 — Example Calling the `WriteEEByte()` Function for a Compiler that Allows M68HC12 Assembly Language to be Inserted Directly into the C Source Code

```
/* C source file showing the necessary M68HC12 assembly language “glue code” */
/* for a C compiler that passes ALL function parameters on the stack. The D-Bug12 callable */
/* functions expect the first function parameter to be passed in the D-accumulator */
/* This example uses the ability of this particular C compiler to insert assembly language */
/* source statements directly into the C source */

/*
        The symbol “Version” should be set to the D-Bug12 version number that is being used.
        For example, if the D-Bug12 version number is 1.0.4, Version should be set to 104.
*/

#define Version 104

#if Version < 200

#define TableBase 0xfe00

#else

#define TableBase 0xf680

#endif

#define mainAddr TableBase + 0x00
#define getcharAddr TableBase + 0x02
#define putcharAddr TableBase + 0x04
#define printfAddr TableBase + 0x06
#define GetCmdLineAddr TableBase + 0x08
#define sscanhexAddr TableBase + 0x0a
#define isxdigitAddr TableBase + 0x0c
#define toupperAddr TableBase + 0x0e
#define isalphaAddr TableBase + 0x10
#define strlenAddr TableBase + 0x12
#define strcpyAddr TableBase + 0x14
#define out2hexAddr TableBase + 0x16
#define out4hexAddr TableBase + 0x18
#define SetUserVectorAddr TableBase + 0x1a
#define WriteEEByteAddr TableBase + 0x1c
#define EraseEEAddr TableBase + 0x1e
#define ReadMemAddr TableBase + 0x20
#define WriteMemAddr TableBase + 0x22

/* The following #defines are used to provide for portability and avoid a linker */
/* conflict with the standard library functions of the same name. No #define is */
/* included for DB12main() since all programs must contain a main() function */

#define printf DB12printf
#define getchar DB12getchar
#define putchar DB12putchar
#define isxdigit DB12isxdigit
#define toupper DB12toupper
#define isalpha DB12isalpha
#define strlen DB12strlen
#define strcpy DB12strcpy

/*****************************************************************************/

Boolean WriteEEByte(Address EEAddress, Byte EEData)

 {

 /* Variable Declarations */


 /* Begin Function WriteEEByte() */

 asm(“puld”);                    /* pull the 'EEAddress' parameter from the stack */
 asm(“jsr [WriteEEByteAddr,pcr]”);/* call D-Bug12's WriteEEByte function */

}       /* end WriteEEByte */

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

6.6 Listing 6 — Example of the D-Bug12 `SetUserVector()` Function to Provide an Interrupt Service Routine that Services a Timer Interrupt

```
;
;
        include  “DB12Macs.Asm”
        opt      lis
;
;
RegBase: equ     $0000
;
TIOS:    equ     RegBase+$80      ; input capture/output compare select.
CFORC:   equ     RegBase+$81      ; force output compare register
OC7M:    equ     RegBase+$82      ; Output compare 7 mask register.
OC7D:    equ     RegBase+$83      ; Output compare 7 data register.
TCNT:    equ     RegBase+$84      ; 16-bit timer/counter register.
TSCR:    equ     RegBase+$86      ; timer system control register.
TQCR:    equ     RegBase+$87      ; Timer Queue control register.
TCTL1:   equ     RegBase+$88      ; Timer control register 1.
TCTL2:   equ     RegBase+$89      ; Timer control register 2.
TCTL3:   equ     RegBase+$8a      ; Timer control register 3.
TCTL4:   equ     RegBase+$8b      ; Timer control register 4.
TMSK1:   equ     RegBase+$8c      ; Main timer interrupt mask register.
TMSK2:   equ     RegBase+$8d      ; Miscellaneous timer interrupt mask register.
TFLG1:   equ     RegBase+$8e      ; Main timer interrupt flag register.
TFLG2:   equ     RegBase+$8f      ; Main Miscellaneous timer interrupt flag register.
TC0:     equ     RegBase+$90      ; Timer input capture/output compare 0.
TC1:     equ     RegBase+$92      ; Timer input capture/output compare 1.
TC2:     equ     RegBase+$94      ; Timer input capture/output compare 2.
TC3:     equ     RegBase+$96      ; Timer input capture/output compare 3.
TC4:     equ     RegBase+$98      ; Timer input capture/output compare 4.
TC5:     equ     RegBase+$9a      ; Timer input capture/output compare 5.
TC6:     equ     RegBase+$9c      ; Timer input capture/output compare 6.
TC7:     equ     RegBase+$9e      ; Timer input capture/output compare 7.
PACTL:   equ     RegBase+$a0      ; Pulse accumulator control register.
PAFLG:   equ     RegBase+$a1      ; Pulse accumulator flag register.
PACNT:   equ     RegBase+$a2      ; 16-bit pulse accumulator count register.
TIMTST:  equ     RegBase+$ad      ; Timer test register.
PORTT:   equ     RegBase+$ae      ; Timer port data register.
PORTTD:  equ     RegBase+$af      ; Timer port data direction register.
;
;
        org      $7000
;
        SetUserVector#UserTimerCh0,#Ch0Int; set the user timer Ch0 interrupt vector.
        ldaa     #$03             ; set the timer prescaler to /8
        staa     TMSK2
        ldaa     #$01
        staa     TIOS             ; set timer Ch0 as an output compare.
        staa     PORTTD           ; set the associated OC0 pin to an output.
        staa     TMSK1            ; enable OC0 interrupts.
        staa     TCTL2            ; set OC0 to toggle on output compares.
        ldd      #5000
        std      TC0              ; set up for a 10 mS period (5 mS each half cycle).
        ldaa     #$90
        staa     TSCR             ; startup the timer/counter system, enable fast clear of interrupt flags.
        cli                       ; enable interrupts.
        bra      *                ; just loop here. the interrupt routine does all the work.
;
;       This timer interrupt routine generates a square wave on port pin PT0
;       using the output compare function of timer channel #0.
;
Ch0Int:  equ     *
        ldd      TC0              ; get the value of the OC0 register.
        addd     #5000            ; add the half period to it.
        std      TC0              ; update the OC0 register automatically clearing the interrupt flag.
        rti
;
```

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor