

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`.

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document. Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor 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 Freescale Semiconductor 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. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor 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 Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor 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 Freescale Semiconductor was negligent regarding the design or manufacture of the part.

