Using and Extending D-Bug12 Routines

By Gordon Doughman

INTRODUCTION

A stable operating environment is required to develop and debug microcontroller software. One of the simplest, most economical development environments consists of a monitor/debugger program that resides in ROM and executes in the target environment.

A ROM monitor provides an environment for controlled execution of software, but cannot perform true emulation because it requires target resources during execution. However, a monitor does provide significant advantages over other debugging environments, because it can provide access to internal utility routines and default exception handlers that would otherwise have to be written by the developer. Resident utility routines can help to provide a stable environment to test new algorithms and conduct performance benchmarks. Default exception handlers can assure graceful recovery if the software under development inadvertently enables peripheral interrupts that do not have associated exception handlers.

This application note provides information that allows a software developer to utilize internal D-Bug12 routines and shows how to substitute user interrupt service routines for default D-Bug12 exception handlers. There are six example listings in the note. Source code is available on-line from Advanced Microcontroller Division Freeware Data Systems:

- BBS — (512) 891- FREE (3733)
- Telnet/FTP — freeware.aus.sps.mot.com
- World Wide Web — http://freeware.aus.sps.mot.com/

UTILITY ROUTINES

D-Bug12 currently provides access to 18 different utility routines through a table of 16-bit pointers beginning at address $FE00, as shown in **Table 1**. See **DESCRIPTION OF D-BUG12 ROUTINES** for details of each routine.

The table of pointers can be extended to $FE7F (128 bytes), allowing access to a maximum of 64 individual utility routines. Using a table of pointers at a fixed address allows access to individual functions to remain constant even though the actual addresses of the routines may be changed when the monitor is modified.

Because D-Bug12 is 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 can also be accessed when programming in assembly language.

Table 1 Utility Routines Summary

Function	Description	Vector Table Address
main()	Start of D-Bug12	$FE00
getchar()	Get a character from SCI0 or SCI1	$FE02
putchar()	Send a character out SCI0 or SCI1	$FE04
printf()	Formatted Output - Translates binary values to characters	$FE06
GetCmdLine()	Obtain a line of input from the user	$FE08
sscanhex()	Convert an ASCII hexadecimal string to a binary integer	$FE0A
isxdigit()	Checks for membership in the set [0..9, a..f, A..F]	$FE0C
toupper()	Converts lower case characters to upper case	$FE0E
isalpha()	Checks for membership in the set [a..z, A..Z]	$FE10
strlen()	Returns the length of a null terminated string	$FE12
strcpy()	Copies a null terminated string	$FE14
out2hex()	Displays 8-bit number as 2 ASCII hex characters	$FE16
out4hex()	Displays 16-bit number as 4 ASCII hex characters	$FE18
SetUserVector()	Setup user interrupt service routine	$FE1A
WriteEEByte()	Write a data byte to on-chip EEPROM	$FE1C
EraseEE()	Bulk erase on-chip EEPROM	$FE1E
ReadMem()	Read data from the M68HC12 memory map	$FE20
WriteMem()	Write data to the M68HC12 memory map	$FE22

FUNCTION CALLING CONVENTIONS

All of the user accessible functions are written in C. In general, parameters are passed to the functions on the stack. All parameters *except the last parameter* must be pushed onto the stack. Parameters are stacked in reverse order from that shown in the C function declaration (right-to-left). The last parameter (the first parameter listed in the function declaration) is passed to the function via the D accumulator. Functions that have only one parameter must pass it in the D accumulator.

Char parameters must always be converted to an int. This means that a parameter declared as a char occupies two bytes of stack space as a parameter. Char parameters must occupy the low order byte (higher byte address) of a word pushed onto the stack. Char parameters must occupy the low order byte of the B accumulator if the parameter is to be 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-bit and 16-bit function results are returned in the D accumulator. Char values returned in the D accumulator are located in the 8-bit B accumulator. Boolean function results are zero for False and non-zero values for True.

Called functions preserve only the content of the stack pointer. If other CPU12 register values must be preserved, they must be pushed onto the stack before any of the parameters and restored after deallocating the parameters.

ASSEMBLY LANGUAGE INTERFACE

Calling a function from assembly language is simple. First, push the parameters onto the stack in the proper order, loading the first or only function parameter into the D accumulator. Then call the function with a JSR instruction. The code following the JSR instruction should remove any parameters pushed onto the stack. If a single parameter is stacked, a PULX or PULY instruction is one of the most efficient ways to unstack it. If two or more parameters are stacked, the LEAS instruction is the most efficient way to remove them. CPU12 registers saved on the stack before the function parameters should be restored with corresponding PUL instructions.

Example

The WriteEEByte() function is called as follows.

```
WriteEEByte:  equu      $FE1C
;
.
.
.
              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 JSR instruction uses a form of indexed indirect addressing that treats 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 a PCR mode. Instead, the PCR mnemonic instructs 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 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 instead.

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

LISTING 1 ASSEMBLY LANGUAGE SOURCE MACROS contains macros that allow routines to be easily called from assembly language. The code was written for the Motorola MCUasm macro assembler. Only slight modification should be required to use the macros with other assemblers. A macro that supports the printf() function is conspicuously absent from the listing. Because printf() accepts a variable number of arguments, it is not possible to construct a macro to easily handle this situation with the Motorola 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 a constant or the address of a 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 USING THE SSCANHEX MACRO** shows how to use a macro.

CALLING D-BUG12 ROUTINES FROM C

Various C compilers pass parameters, return function results, and deallocate local variables and parameters differently. This makes accessing D-Bug12 functions from C 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 of effort is required. The header file shown in **LISTING 3 STANDARD FUNCTION LIBRARY NAMES** can be #included with any source file that references the 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 helps to ensure portability of the program text. In addition, using the C preprocessor to replace the standard function library names with names prefixed by "DB12" allows a program to use other functions in a standard function library without creating duplicate function conflicts in the linker. **LISTING 4 GETCMDLINE( ) AND PRINTF( ) FUNCTIONS** shows how these functions 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" must be written for each of the D-Bug12 user accessible functions. The size and complexity of the code depends upon how closely the compiler follows the D-Bug12 function calling convention.

If, for example, a compiler passes all of its parameters on the stack rather than passing the first parameter in the D accumulator, it would be necessary to pull the first parameter from the stack into the D accumulator first, then execute a JSR instruction to call the D-Bug12 function. **LISTING 5 CALLING THE WRITEEE-BYTE() FUNCTION** is an example of calling a function in 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 must be assembled into an object file and combined with the compiled C source code with the linker.

INTERRUPT SERVICE ROUTINES

One of the advantages of D-Bug12 is its ability to provide default exception (interrupt) handlers. These default exception handlers can help to provide graceful recovery if software inadvertently enables peripheral interrupts. However, most developers will provide peripheral interrupt service handlers as part of application development. The D-Bug12 SetUserVector() function allows software developers to substitute custom interrupt service routines for any of the default D-Bug12 exception handlers.

D-Bug12 accesses user interrupt service routines through a RAM-based interrupt vector table that mirrors the CPU12 interrupt vectors. These vectors are located in EPROM, in addresses $FC00 to $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 interrupt service routine. If the corresponding RAM interrupt vector table entry contains an address of $0000, control is returned to the D-Bug12 monitor where an exception message and CPU register contents are displayed. The maximum frequency at which interrupts can occur is slightly lower than when code is run from EPROM, due to the small amount of additional code D-Bug12 must execute to determine whether a user interrupt service routine is to be called.

Interrupt service routines can consist of any number of CPU12 instructions, but must end with the RTI (Return from Interrupt) instruction, which causes normal program execution to resume. However, the interrupt service request that invoked the service routine must be cleared before RTI is executed. This is generally accomplished by writing to the control registers of the on-chip peripheral that is the source of the request. If a service request is not cleared before return, the CPU12 will get "stuck" in the interrupt service routine and repeat it until the system is reset.

LISTING 6 USING THE SETUSERVECTOR() FUNCTION shows how to use the function to provide an interrupt service routine that services a timer interrupt.

DESCRIPTION OF D-BUG12 ROUTINES

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

void main(void);

    Pointer Address: $FE00

    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() changes. Because the user can replace the supplied startup routines with custom startup code, the supplied D-Bug12 startup object code must be examined to determine the address of main().

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

```
jmp     [$fe00,pcr]
```

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 reasons discussed above, providing a pointer to the main() function at a fixed address allows the location of main() to change without changing the code.

When a user program stored in on-chip EEPROM is executed from power up or reset, D-Bug12 should be entered via the startup code at the label "DEBUG12" rather than the main() function, because main() does not perform hardware initialization and does not clear variable memory.

Do not use the main() function to re-enter D-Bug12 from a program that began execution from D-Bug12. Using main() reinitializes D-Bug12 tables and variables. Previously set breakpoints are lost and breakpoint SWIs remain in the users program. The proper way to re-enter D-Bug12 under program control is to place an SWI in the user program. When this SWI is executed, D-Bug12 performs the same actions it performs for its own breakpoint SWI.

int getchar(void);

    Pointer Address: $FE02

    Stack Space: 2 bytes

The getchar() function retrieves a single character from the control terminal SCI. If a character is not available in the SCI Receive Data Register when the function is called, getchar() waits until one is received. Because the character is returned as an int, the 8-bit character is placed in the B accumulator.

int putchar(int);

    Pointer Address: $FE04

    Stack Space: 4 bytes

The putchar() function provides the ability to send a single character to the control terminal SCI. If the SCI Transmit Data Register is full when the function is called, putchar() waits 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 error conditions that occur in the process and therefore never returns the EOF value (-1). Because the character is returned as an int, the 8-bit character is placed in the B accumulator.

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

    Pointer Address: $FE06

    Stack Space: Minimum of 64 bytes, does not include parameter stack space.

The printf() function is used to convert, format, and print its arguments on the standard output under the 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 are copied directly from the format string to the display device; conversion specifications 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]

Character Description

- (minus sign)   Left justifies the converted argument.

FieldWidth   Integer number that specifies the minimum field width for the converted argument. The argument is displayed in a field at least this wide. The displayed argument is 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 causes the value of the next argument in the argument list to be used instead.

Table 2 shows 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 above or the conversion characters shown in Table 2 below, the behavior of the printf() function is undefined.

Table 2 Printf() Conversion Characters

Character	Argument Type; Displayed As
d, i	int; signed decimal number
o	int; unsigned octal number (without a leading 0)
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.

Example 1

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

Where Num has the value $FFFF

Displays the result:

    Signed Decimal: -1 Unsigned Decimal: 65535

Example 2

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

Where Num has the value $FF

Displays the result:

    Hexadecimal: FF Hexadecimal: 00FF

Example 3

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

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

Displays the result:

    This is a Test

Notice that the formatting string in the above examples includes both line feed (\n) and carriage return (\r) characters. The D-Bug12 printf() function does not automatically send a carriage return to the display device when a line feed appears in the formatting string.

int GetCmdLine(char *CmdLineStr, int CmdLineLen);

    Pointer Address: $FE08
    Stack Space: 11 bytes

The GetCmdLine() function is used to obtain a line of input from the user. GetCmdLine() accepts input from the user, one character at a time, by calling getchar(). As each character is received, it is echoed back to the terminal by calling putchar() and placed in the character array pointed to by CmdLineStr. A maximum of CmdLineLen - 1 printable characters can 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 echoes the backspace to the terminal, prints the ASCII space character ($20), and sends a second backspace character to the terminal. This causes 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 (0).

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

    Pointer Address: $FE0A
    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 can 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.

int isxdigit(int c);

Pointer Address: $FE0C
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.

int toupper(int c);

Pointer Address: $FE0E
Stack Space: 4 bytes

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

int isalpha(int c);

Pointer Address: $FE10
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.

unsigned int strlen(const char *cs);

Pointer Address: $FE12
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 an ASCII Null ($00) character.

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

Pointer Address: $FE14
Stack Space: 8 bytes

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

void out2hex(unsigned int num);

Pointer Address: $FE16
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 who cannot use the printf() function. Out2hex() simply calls printf() with a format string of "%2.2X".

void out4hex(unsigned int num);

    Pointer Address: $FE18

    Stack Space: 70 bytes

Out4hex() displays num on the control terminal as four hexadecimal characters.This function is provided for those who cannot use the printf() function. Out4hex() simply calls printf() with a format string of "%4.4X".

int SetUserVector (int VectNum, Address UserAddress);

    Pointer Address: $FE1A

    Stack Space: 8 bytes

The function SetUserVector() allows substitution of user 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 can be changed without having to change a users code. The address of the interrupt service routine, passed in UserAddress, should point to a routine that ends with an M68HC12 RTI instruction.

The following example defines the valid constants for VectNum. If an invalid constant is passed in VectNum, SetUserVector() returns a value of -1, otherwise a value of zero is returned.

Example

```
typedef Address char *;
typedef Byte unsigned char;
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 };
```

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

Passing the constant 'RAMVectAddr' in the VectNum parameter causes the base address of the RAM interrupt vector table to be returned instead of an error code. This allows 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.

Use care 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 can jump to an incorrect program address, resulting in loss of CPU/monitor control.

Boolean WriteEEByte(Address EEAddress, Byte EEData);

Pointer Address: $FE1C
Stack Space: 12 bytes

The WriteEEByte() 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 users 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, false (zero value) is returned by the function.

int EraseEE(void);

Pointer Address: $FE1E
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, a check of 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.

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

Pointer Address: $FE20
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 program 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.

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

Pointer Address: $FE22
Stack Space: 22 bytes

The WriteMem() function is used internally by D-Bug12 for all memory write accesses. WriteMem() is different from ReadMem() in the fact 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.

LISTING 1 ASSEMBLY LANGUAGE SOURCE MACROS

```
;
;constants used with the SetUserVector() function to set the address of user supplied
;interrupt service routines.
;
UserPortHKWU: equ             7
UserPortJKWU: equ             8
UserAtoD:     equ             9
UserSCI1:     equ             10
UserSCI0:     equ             11
UserSPI0:     equ             12
UserTimerCh0: equ             13
UserTimerCh1: equ             14
UserTimerCh2: equ             15
UserTimerCh3: equ             16
UserTimerCh4: equ             17
UserTimerCh5: equ             18
UserTimerCh6: equ             19
UserTimerCh7: equ             20
UserPAccOvf:  equ             21
UserPAccEdge: equ             22
UserTimerOvf: equ             23
UserRTI:      equ             24
UserIRQ:      equ             25
UserXIRQ:     equ             26
UserSWI:      equ             27
UserTrap:     equ             28
RAMVectAddr:  equ             -1              ; returns base address of vector table.
;
; C function: void main(void);
;
main:         macro
              jmp             [$fe00,pcr]  ; restart D-Bug12 from main().
              endm
;
; C function: int getchar(void);
;
getchar:      macro
              jsr             [$fe02,pcr]  ; call getchar routine. return cha in B.
              endm
;;
; C function: int putchar(int);
;
putchar:      macro
              ldab            \1              ; load the character to send into B.
              jsr             [$fe04,pcr]  ; call getchar routine. return sent char in B.
              endm
;
; C function: int GetCmdLine(char *CmdLineStr, int CmdLineLen);
;
GetCmdLine:   macro
              ldd             \2              ; load the length of the command line
                                              ; character buffer.
              pshd                            ; place it on the stack.
              ldd             \1              ; get a pointer to the character buffer
              jsr             [$fe08,pcr]  ; go get characters from the user.
              pulx                            ; remove command line length param from stack.
              endm
;
```

```
; C function: char * sscanhex(char *HexStr, unsigned int *BinNum);
;
sscanhex:    macro
             ldd            \2             ; get pointer to location for result.
             pshd                          ; place it on the stack.
             ldd            \1             ; get pointer to ASCII hex string to convert.
             jsr            [$fe0a,pcr]    ; convert ASCII hex string to binary
             pulx                          ; one byte instruction to remove the pointer
                                           ; to the conversion result from the stack.
             endm
;
; C function: int isxdigit(int c);
;
isxdigit:    macro
             ldab           \1             ; load ASCII character into the B accumulator.
             jsr            [$fe0c,pcr]    ; 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            [$fe0e,pcr]    ; convert the character to upper case if the
                                           ; character is in the set a..z.
             endm
;
; C function: int isalpha(int c);
;
isalpha:     macro
             ldab           \1             ; load ASCII character into the B accumulator.
             jsr            [$fe10,pcr]    ; 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            [$fe12,pcr]    ; count number of characters in the string.
             endm
;
; C function: char * strcpy(char *s1, char *s2);
;
strcpy:      macro
             ldd            \2             ; get source string pointer(s2) onto stack.
             pshd                          ; place it on the stack.
             ldd            \1             ; get pointer to destination string (s1)
             jsr            [$fe14,pcr]    ; 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            [$fe16,pcr]    ; 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             [$fe18,pcr]     ; display the word.
                endm
;
; C function: int SetUserVector (int VectNum, Address UserAddress);
;
SetUserVector:macro
                ldd             \2              ; get address of interrupt service routine.
                pshd                            ; place it on the stack.
                ldd             \1              ; get the interrupt vector to set.
                jsr             [$fe1a,pcr]     ; set the user's interrupt vector.
                pulx                            ; one byte instruction to remove address of
                                                ; 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             [$fe1c,pcr]     ; program the EEPROM byte.
                pulx                            ; one byte instruction to remove EEPROM data
                                                ; from the stack.
                endm
;
; C function: int EraseEE(void);
;
EraseEE:        macro
                jsr             [$fe1e,pcr]     ; 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 pointer to buffer for memory contents.
                pshd                            ; place it on the stack.
                ldd             \1              ; get address to start reading data.
                jsr             [$fe20,pcr]     ; read data from target memory.
                leas            4,s             ; remove the 2 parameters placed on the stack.
                endm
;
; C function: int WriteMem (Address StartAddress, Byte *MemDataP, unsigned int NumBytes);
;
WriteMem:       macro
                ldd             \3              ; get the number of bytes to write.
                pshd                            ; place it on the stack.
                ldd             \2              ; get pointer to data to be placed in memory.
                pshd                            ; place it on the stack.
                ldd             \1              ; get starting address for data.
                jsr             [$fe22,pcr]     ; write data into the target memory.
                leas            4,s             ; remove the 2 parameters placed on the stack.
                endm
;
```

LISTING 2 USING THE SSCANHEX MACRO

This program demonstrates several of the D-Bug12 functions. 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 a number entered in hexadecimal in signed decimal, hexadecimal, and unsigned decimal formats.

5. Returns to D-Bug12 when a blank line is entered.

```
;
            include     "DB12Macros.Asm"
            opt         lis
;
Space       equ         $20                             ; space character.
;
            org         $800
Buffer      ds          20                              ; character buffer for user input
BufferP     ds          2                               ; pointer into character buffer.
BinNum      ds          2                               ; converted hex number.
;
;
            org         $7000
;
Test:       ldd         #PromptStr                      ; load pointer to prompt string.
            jsr         [printf,pcr]                    ; print the prompt.
            GetCmdLine  #Buffer,#20                     ; get an ASCII hex number.
            ldx         #Buffer                         ; point to start of buffer.
SkipSpcs:   ldaa        0,x                             ; get character from buffer.
            cmpa        #Space                          ; leading space character?
            bne         DoneSkip                        ; no, done.
            inx                                         ; yes, point to next character
                                                        ; in buffer.
            bra         SkipSpcs                        ; check for another space.
DoneSkip:   stx         BufferP                         ; save pointer to first
                                                        ; non-blank character in buffer.
            tst         0,x                             ; check for blank line.
            beq         Done                            ; return to D-Bug12 if done.
            sscanhex    BufferP,#BinNum                 ; convert from ascii to binary.
            cpd         #0
            bne         PrtResult
            ldd         #Error                          ; load pointer to error string.
            jsr         [printf,pcr]                    ; print the error message.
            bra         Test
PrtResult:  ldd         BinNum                          ; get value of converted number.
            pshd                                        ; place three copies of number
                                                        ; on 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 decimal = %u",$0d,
                        $0a,0
```

LISTING 3 STANDARD FUNCTION LIBRARY NAMES

This file can be included with any C source file that uses D-Bug12 user callable routines. It provides a simple, portable way to access the routines from C without having to using assembly language "glue code"

```c
/* some typedefs used by D-Bug12 */

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

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 };

/* structure that defines the functions in D-Bug12's user accessable */
/* function table. Also provides a function prototype for each function */
/* Documentation for each of these functions can be found in Application */
/* Note AN1280 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 accessable functable */

#define DBug12FNP  ((UserFNP)0xfe00)

/* 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
```

LISTING 4 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 */
```

LISTING 5 CALLING THE WRITEEEBYTE() FUNCTION

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.

```
#define mainAddr 0xfe00
#define getcharAddr 0xfe02
#define putcharAddr 0xfe04
#define printfAddr 0xfe06
#define GetCmdLineAddr 0xfe08
#define sscanhexAddr 0xfe0a
#define isxdigitAddr 0xfe0c
#define toupperAddr 0xfe0e
#define isalphaAddr 0xfe10
#define strlenAddr 0xfe12
#define strcpyAddr 0xfe14
#define out2hexAddr 0xfe16
#define out4hexAddr 0xfe18
#define SetUserVectorAddr 0xfe1a
#define WriteEEByteAddr 0xfe1c
#define EraseEEAddr 0xfe1e
#define ReadMemAddr 0xfe20
#define WriteMemAddr 0xfe22

/* 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 'EEAddress' parameter from stack */
  asm("jsr [WriteEEByteAddr,pcr]");     /* call WriteEEByte function */

  }/* end WriteEEByte */
```

LISTING 6 USING THE SETUSERVECTOR() FUNCTION

```
                include       "DB12Macros.Asm"
;
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    ; Misc timer interrupt mask register.
TFLG1:          equ           RegBase+$8e    ; Main timer interrupt flag register.
TFLG2:          equ           RegBase+$8f    ; Main Misc 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 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 10 mS period (5 mS per half cycle).
                ldaa          #$90
                staa          TSCR           ; startup the timer/counter system, enable
                                               fast clear of interrupt flags.
                cli                          ; enable interrupts.
                bra           *              ; Loop. The interrupt routine does 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, clr int flag.
                rti
;
```

Ⓜ *MOTOROLA*