dBUG FIRMWARE DEVELOPMENT

dBUG is a traditional ROM monitor/debugger designed for rapid platform bring-up. By providing a small number of well-defined functions, dBUG can be fitted to a new hardware platform very quickly.

1. Overview

Written in the C language, dBUG is a portable debugger engineered for systems designed around Motorola's processor architectures. dBUG provides a common debugging interface for all these hardware systems. To accomplish this, dBUG is modularized into three components:

User interface component
CPU-specific component
Board-specific component

The user interface component consists of a set of standard commands that provide basic debugging facilities. These commands are the same on all systems.

The CPU-specific component implements all details and services specific to the processor. The user interface and board-specific components draw upon resources provided by the CPU-specific component.

The board-specific component implements all the remaining services required by the user interface and CPU-specific components. These services include platform initialization and basic character input and output. The board-specific component also implements additional commands and features that are required by the particular system.

The steps needed to build existing and create new board support packages (BSP) for dBUG are detailed in the sections below.

2. Directory Structure

The dBUG source tree is depicted here:

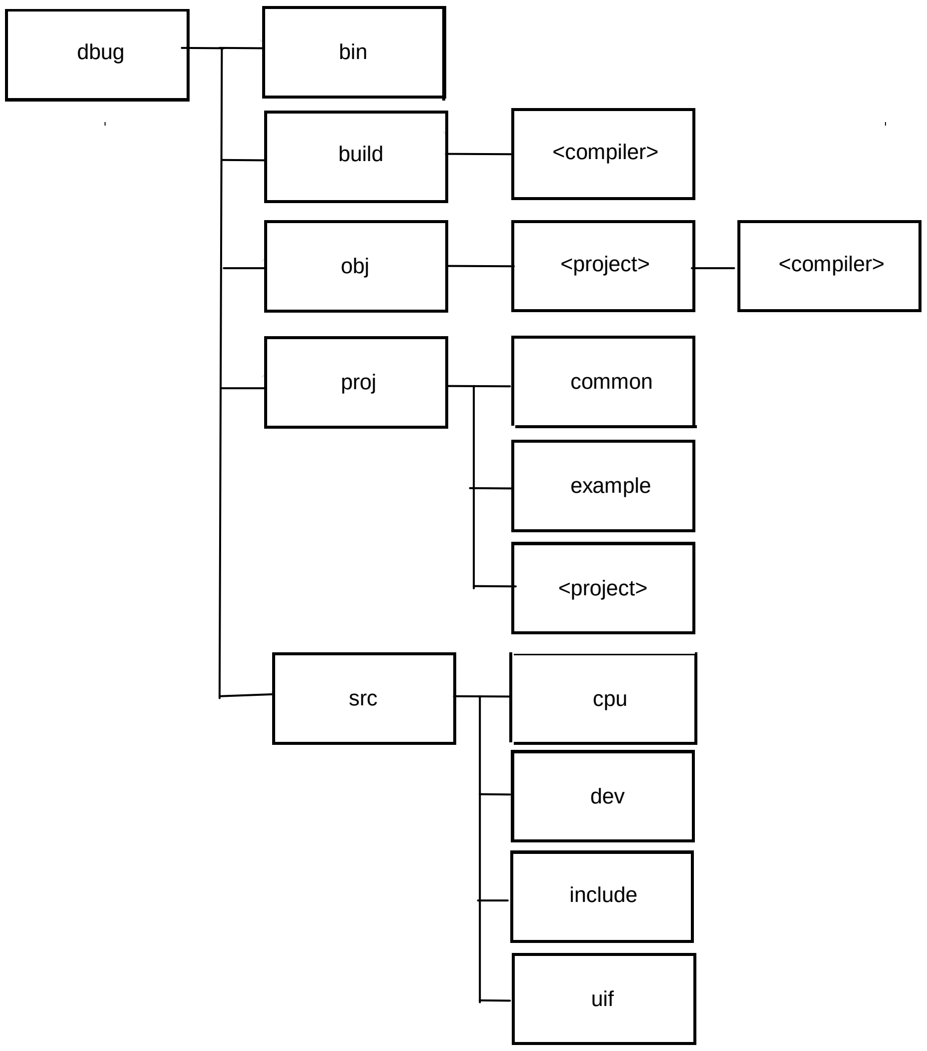

Figure 2-1 dBUG Source Tree

Located in *dbug/* is the top-level makefile for building dBUG with command line toolchains (diab, gnu, etc.). The top-level Makefile invokes a subordinate makefile that performs the actual work. Each project contains its own makefile and linker files.

The directory *dbug/bin* contains miscellaneous tools for building projects.

The directories under *dbug/build/* contain compiler specific build settings including pointers to compiler, linker, and other build utilities.

The *dbug/obj* directory holds the output object and executable files. This directory is created during the build process.

The directory *dbug/src/cpu* contains the source to the CPU-specific component. An entire subdirectory structure and source code base exists for the various processors supported by dBUG.

The directories under *dbug/src/dev* contain device drivers used by several of the board projects.

The directories under *dbug/src/include* contain C header files for dBUG as well as processor header files.

The directory *dbug/src/uif* contains the source to the user interface component. The *dbug/src/uif/net* directory contains the networking source files.

The directory *dbug/proj* is where the source for the various BSPs is located. Each BSP is called a project, in dBUG terminology. Under this directory, each project has its own subdirectory. Projects must be located here, as all paths to source files are relative to the project subdirectory.

The subdirectory structure under each board project is depicted here:

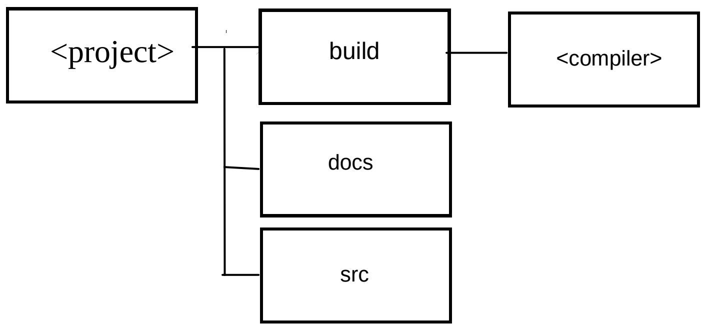

Figure 2-2 Project Directory Structure

The directory *<project>/src/* contains the board-specific C files.

The directory *<project>/build/* contains the host toolchain specific files. Typically these files are makefiles, linker script files and any toolchain specific assembly source files. The directory *<compiler>* will reflect the toolchain in use; for example, diab, or mwerks. If command line tools are used, the top level makefile, *dbug/Makefile*, calls the subordinate makefile located in this directory. If GUI build tools are used, their project build files are located in this directory.

3. Building a dBUG Project

3.1 Environment Setup

The dBUG build system (except for GUI based compilers) requires a Unix-like environment. If you are developing on a Win32 machine, you will need to install a software package such as Cygwin.
The Cygwin tools are ports of the popular GNU development tools and utilities for all modern versions of Windows. They function by using the Cygwin library (cygwin1.dll), which provides a UNIX-like API on top of the Win32 API.

You can obtain the Cygwin tools from http://sources.redhat.com/cygwin/.

3.2 Compiler Support

The dBUG Firmware development system currently supports the Metrowerks CodeWarrior and Diab Data compiler suites. However, the source is written so that it is very compiler independent and capable of being easily ported to other build systems.

3.2.1 Diab Data

The Diab toolchain requires a command line environment. The top level Makefile is located in the *dbug/* directory. The top-level Makefile invokes a subordinate makefile that performs the actual work. Each project contains its own makefile and linker files (located in *dbug/proj/<project>/build/<compiler>*). The makefiles use pointers to the Diab utilities which are defined in *dbug/build/diab/<cpu>.comp*. These pointers must be set appropriately before attempting to build a project.
To build a project, execute the top-level makefile with the appropriate project as an argument (i.e. “make m5282evb”).

3.2.2 Metrowerks CodeWarrior

Metrowerks CodeWarrior build projects (.mcp files) have been set up for some of the dBUG projects. dBUG projects with CodeWarrior support will have a <project>.mcp file in the *dbug/proj/<project>/build/mwerks* directory.

4. Files in the Board Support Package

A number of files are needed to complete a board support package. Source files are needed for the BSP itself, as well as makefiles and linker script files. The following files typically exist for a dBUG project:

- *build.c*
- *config.h*
- *evbcmds.c*
- *<project>.c*
- *<project>.h*
- *sysinit.c*

File *build.c* is used to keep track of a build number and build time and date.

File *config.h* is required to contain one item: the type of processor. A #define identifies the CPU in use, which in turn directly affects which header files are automatically included. This file may optionally #define DBUG_NETWORK to indicate that the TFTP network download capability is to be available.

File *evbcmds.c* contains the dBUG command set as well as the SET/SHOW options.

File <project>.c contains the board-specific routines that are required by dBUG. Information about these routines is provided in Section 9 Board-Specific Functions.

File *<project>.h* contains configuration information, definitions and prototypes specific to the platform. Normally this file is #included by *config.h* to make it visible to all files in the BSP.

File *sysinit.c* provides reset configuration for the processor/platform.

Other board-specific files for drivers, diagnostics or commands are located here as well. Some device drivers may also be in the *dbug/src/dev* directory.

The toolchain specific files are isolated into the *dbug/proj/<project>/build/<compiler>* subdirectory. Typically the files located here are the subordinate makefile or compiler-specific project, linker scripts, and board-specific system calls (compiler specific assembly files).

Files common to several BSPs are kept in the *dbug/proj/common* directory.

5. Creating a Board Support Package

The first stage of creating the BSP is to do the minimum work necessary to allow dBUG to boot. Once dBUG is able to boot, features can be incrementally added.

5.1 Board Support Package Template Files

Accompanying dBUG is the source to a generic board support package which contains all the basic files and functions needed for a dBUG port. It is recommended that the generic board support package be copied into a new project directory as the basis for the new dBUG port. In most cases, the generic templates need only be completed with the board-specific details. The generic board support package can be found in the *dbug/proj/example* directory.

Alternatively, a completed dBUG project with similar features may serve well as a starting point.

5.2 Initial Board Support Package

The steps for completing the initial board support package are straightforward.

1. Edit *config.h* and define the appropriate CPU. See *dbug/src/include/cpu/cpu.h* for a complete list of supported processors. Do NOT define DBUG_NETWORK at this time.
2. Edit *<project>.h* to provide any necessary prototypes, data structures or definitions. Memory map and device definitions provided in this file often prove useful.
3. Edit *<project>.c* and provide the details to the required board-specific functions.
4. Create any required CPU-specific functions in *sysinit.c*. For example, the integrated processors supported by dBUG require board-specific initialization of the integrated peripherals.
5. Edit the makefiles or compiler project files and the linker scripts in the *build/<compiler>* directory to accommodate the toolchain in use. Exact details for configuring the toolchain and linker files are beyond the scope of this document, and must be referred to the host toolchain documentation. Be careful to place ROM and RAM sections correctly!
6. If using command line tools, modify the top-level project makefile to invoke the correct subordinate makefile and define the correct output directory.

Upon completing the above steps, the project can be built as indicated in Section 3 Building a dBUG Project.

6. Debugging a Board Support Package

After building the BSP, some debugging may be necessary. The two most problematic areas requiring debug are the initialization code and toolchain related issues.

6.1 dBUG Run-Time Entry Points

To aid the debugging of initialization code, it is useful to know the execution path of dBUG out of reset. The execution path at reset performs all basic initialization of the system.

The reset execution path as well as the other run-time execution paths is detailed below.

dBUG obtains control at three primary entry points:

- Reset
- General Exception (excluding interrupts)
- Interrupts

The Reset entry point is executed at board power-up, board hard reset, or the RESET command. The general code sequence executed is the following:

1. reset vector - The reset vector, located in *dbug/src/cpu/<cpu_family>/vectors.s*, points to asm_startmeup.
2. asm_startmeup - Located in *dbug/src/cpu/<cpu_family>/<processor>_lo.s*, this code invalidates caches, disables interrupt, caching and address translation, and sets other CPU internal resources to a disabled, known state. Depending upon the processor, CPU-specific initialization code in the board support package is executed. When complete, main() is invoked.
3. main() - Located in *dbug/src/uif/main.c*, this routine performs the remaining initialization of the board and dBUG. This routine copies the vector table from ROM to RAM, copies initialized data (.data section) from ROM to RAM, and zeroes uninitialized data (.bss section). The following functions are then called, in sequence: board_init(), cpu_init(), uif_init(), board_init2(), uif_cmd_ver(), board_init3(), and finally mainloop().
4. board_init() - Located in *dbug/proj/<project>/src/<project>.c*, this routine, at a minimum, initializes the dBUG console port.
5. cpu_init() - Located in *dbug* Located in *dbug/src/cpu/<cpu_family>/<processor>_hi.c*, this routine initializes the CPU specific internal variables and resources.
6. uif_init() - Located in *dbug/src/uif/cmds.c*, this routine performs the initialization of dBUG internal variables and resources.
7. board_init2() - Located in *dbug/proj/<project>/src/<project>.c*, this routine performs activities that require dBUG resources (such as registering an interrupt handler), or activities prior to displaying the dBUG banner (such as displaying the amount of installed memory).
8. uif_cmd_ver() - Located in *dbug/src/uif/cmds.c*, this function displays the dBUG version banner.
9. board_init3() - Located in *dbug/proj/<project>/src/<project>.c*, this routine performs any activities prior to entering the interactive dBUG> command prompt (such as booting an operating system or other system software).
10. mainloop() - Located in *dbug/src/uif/main.c*, this routine enters into an infinite loop which displays the dBUG> command prompt and processes user input.

The General Exception entry point is encountered during memory access errors, breakpoints, single instruction tracing, and other general exceptions. The general code sequence executed is the following:

1. exception vector - The vector, located in *dbug/src/cpu/<cpu_family>/<processor>/vectors.s*, points to asm_exception_handler.

2. asm_exception_handler - Located in , this code flushes and disables caches, disables interrupts and address translation, and stores the register context. When complete, cpu_handler() is invoked with the exception number.
3. cpu_handler() - Located in *dbug/src/cpu/<cpu_family>/<processor>_hi.c*, this routine handles the exception. In most cases, the exception number and context information is displayed, and control passed to mainloop(). However, some exceptions (software breakpoints, for example) may return from cpu_handler() to asm_exception_handler, at which point the context is restored and execution resumes.

The Interrupt entry point is executed upon detection of a CPU interrupt. These interrupts are generated primarily by peripheral devices and require servicing. The general code sequence executed is the following:

1. interrupt vector - The vector, located in *dbug/src/cpu/<cpu_family>/vectors.s*, points to asm_isr_handler.
2. asm_isr_handler - Located in *dbug/src/cpu/<cpu_family>/<processor>_lo.s*, this code saves volatile registers (as per the calling convention/ABI) on the current stack, and invokes isr_execute_handler() with the interrupt number.
3. isr_execute_handler() - Located in *dbug/src/uif/isr.c*, this routine searches the table of interrupt service routines (ISRs) registered with dBUG. If a match is located, the ISR is invoked, and its return value (TRUE or FALSE) is returned to asm_irq_handler. If no match is found, FALSE is returned to asm_irq_handler.
4. If the return value from isr_execute_handler() is TRUE, indicating the interrupt was serviced, then the context is restored and execution resumes. If the return value is FALSE, the complete register context is saved, cpu_handler() is invoked to display the exception information, and control passed to mainloop().

If user code takes over any of these entry points, then it is quite possible that dBUG will not work properly, if at all.

6.2 Compiler/Toolchain Considerations

An important toolchain issue to understand is the run-time memory footprint. For most systems, dBUG executes from ROM or Flash memory, and uses RAM starting at address 0x00000000. Table illustrates the typical memory footprint for these systems.

Table 6-1 dBUG Run-Time Memory Footprint

SYMBOL	MEMORY SECTION	COMMENT
__DATA_ROM	.data	dBUG's initialized data is stored in ROM but copied to RAM at boot-time
	.text	The executable code for dBUG
__USER_SPACE	USER RAM	This portion of memory is deemed usable by user programs.
__SP_INIT __SP_END	STACK	Stack space for dBUG
__HEAP_END __HEAP_START	HEAP	Heap space for dBUG
__BSS_END __BSS_START	.bss	dBUG's uninitialized data. It is cleared to zero at boot-time
__DATA_END __DATA_RAM	.data	dBUG's initialized data. It is copied from ROM to RAM at boot-time.
__VECTOR_RAM	VECTOR TABLE	CPU vector table for dBUG, normally located at 0x0000_0000

The .text section contains the executable code for dBUG. The .data section contains initialized data which is stored in ROM, but copied to RAM at boot-time. The .bss section is the uninitialized data for dBUG that is zeroed at boot-time.

The symbol names in the left-hand column are defined in the linker script file, and evaluate to a 32-bit value representing the appropriate address. For example, __DATA_RAM is the address of the .data section in RAM.

In most systems, dBUG requires 128K of ROM and 64K of RAM. The ROM provides storage for dBUG's executable code and initialized data, while RAM contains a CPU vector table, the run-time initialized and uninitialized data, heap, and stack space. The actual amount of ROM and RAM required will vary depending upon the features added to dBUG as well as the compiler in use.

To conserve RAM used by dBUG, declare all constant data with the C qualifier *const* or *static const*. Constant initialized data will remain in ROM, thus not requiring any RAM at run-time.

To take advantage of the CPU architecture, compilers may produce sections other than .text, .data, and .bss. The advantage of additional sections is that references to items located in these sections are normally very quick; requiring a single instruction with an embedded offset to access the item. The disadvantage is that at all times one or more CPU registers must contain a pointer to the additional sections; this poses a problem for dBUG. By the nature of the environment, dBUG cannot provide any protection between itself and downloaded code, thus the values of CPU registers can be changed at any time by the downloaded code. Therefore, to support additional sections, the CPU registers must be managed at every possible entry point into dBUG (entry points are not just exceptions, but include system calls, interrupt handlers and functions directly callable by downloaded code). While it is possible to provide the necessary management, this significantly increases the complexity of dBUG, while not necessarily guaranteeing its reliability. In short, limit the compiler-generated sections to .text, .data, and .bss sections; items located in these sections are referenced by the compiler with absolute addresses.

When interfacing assembly routines with C routines, the appropriate application binary interface (ABI) must be used. The ABI defines the usage of CPU registers and how parameters are passed between functions. In general, dBUG uses the ABI as defined by System V Release 4 Unix, SVR4. Also, most toolchains use differing and incompatible assembly source file formats, which add to the difficulty of using assembly source files.

The placement of the dBUG vector table is an important toolchain issue. The linker must place the dBUG vector table so that the CPU can access it at power-on reset. File *vectors.s* contains the dBUG vector table and should be placed first by the linker.

7. Adding Features to the Board Support Package

Once the basic BSP is working, features and new commands can easily be added to dBUG.

7.1 Adding Commands

dBUG provides a core set of commands for performing basic system debugging activities. The command set can be extended to suit the particular board or application needs.

When the user enters a command, two searches through the command table are performed in order to locate the command. The first search seeks an exact match on the user-specified command and a command name in the table. If this search fails, a second search is performed seeking a match on the shortened command names.

The board-specific file *dbug/proj/<project>/src/evbcmds.c* contains the dBUG command table.

```
UIF_CMD UIF_CMDTAB[] =
{
    UIF_CMDS_ALL
    CPU_CMDS_ALL
};
const int UIF_NUM_CMD = UIF_CMDTAB_SIZE;
```

The core command set is inserted with the macro UIF_CMDS_ALL, defined in dbug.h, and any CPU-specific commands are inserted with the macro CPU_CMDS_ALL. Additional commands are placed in the table following these macros. File *dbug/src/include/dbug.h* defines the command table entry data structure.

```
typedef const struct
{
      char *   cmd;          /* command name user types, i.e. GO   */
      int      unique;       /* num chars to uniquely match       */
      int      min_args;     /* min num of args command accepts   */
      int      max_args;     /* max num of args command accepts   */
      int      flags;        /* command flags (repeat, hidden)    */
      void     (*func)(int, char **); /* actual function to call  */
      char *   description;  /* brief description of command      */
      char *   syntax;       /* syntax of command                 */
} UIF_CMD;
```

Field *cmd* is the command name as it is typed on the command line. Command names are eight characters or less in length.

Field *unique* indicates the number of characters required to match for the short name of the command. This value must be greater than zero, and less than the length of the command name.

Field *min_args* indicates the minimum number of arguments the command requires. If the user specifies fewer arguments than this field indicates, an error message is produced and the command is not invoked. This field must be equal to or greater than zero.

Field *max_args* indicates the maximum number of arguments the command accepts. If the user specifies more arguments than this field indicates, an error message is produced and the command is not invoked. The value for this field must equal or exceed the value for min_args, and may not exceed UIF_MAX_ARGS.

Field *flags* is used to slightly modify the behavior of the command. Flag UIF_CMD_FLAG_REPEAT indicates that the command is capable of rapid repeat execution. This flag indicates that the user may enter the command once, and then press <Return> to invoke subsequent executions of this command, i.e. the TRACE command. Flag UIF_CMD_FLAG_HIDDEN prevents the command from being displayed in the HELP menu.

Field *func* is the function to invoke when a command line matches the command name and meets its argument requirements. Function func receives two arguments, the first is the number of tokens on the command line (there is always at least one: the command), and the second is a pointer to an array of pointers pointing to each token on the command line. This scheme is similar to the invocation of the C language main() function.

Field *description* is the verbal description of the command displayed in the HELP menu.

Finally, field *syntax* describes the command usage and options. This information is displayed in the HELP menu.

For examples, the core command entries are located in *dbug/src/include/dbug.h*.

7.2 Adding SET/SHOW Options

dBUG provides a core set of SET/SHOW options for configuring dBUG. The SET/SHOW option set can be extended to suit the particular needs of the board.

When the user enters the SHOW command, the setting for the particular option is displayed. If no option is specified, then all option values are displayed.

When the user enters the SET or SHOW command, two searches through the SET/SHOW option table are performed in order to locate the option. The first search seeks an exact match on the user-specified option and an option name in the table. If this search fails, a second search is performed seeking a match on the shortened option names.

The board-specific file *dbug/proj/<project>/src/evbcmds.c* contains the dBUG SET/SHOW option table.

```
UIF_SETCMD UIF_SETCMDTAB[] =
{
    UIF_SETCMDS_ALL
    CPU_SETCMDS_ALL
};
const int UIF_NUM_SETCMD = UIF_SETCMDTAB_SIZE;
```

The core option set is inserted with the macro UIF_SETCMDS_ALL, defined in dbug.h, and any CPU-specific commands are inserted with the macro CPU_SETCMDS_ALL. Additional options are placed in the table following these macros. File *dbug/src/include/dbug.h* defines the option table entry data structure.

```
typedef const struct
{
    char *  option;
    int     unique;
    int     min_args;
    int     max_args;
    int     flags;
    void    (*func)(int, char **);
    char *  syntax;
} UIF_SETCMD;
```

Field *option* is the option name as it is typed on the SET/SHOW command line. Option names are eight characters or less in length.

Field *unique* indicates the number of characters required to match for the short name of the option. This value must be greater than zero, and less than the length of the option name.

Field *min_args* indicates the minimum number of arguments the option requires. The value for this field must be at least one. If the user specifies fewer arguments than this field indicates, an error message is produced. This field must be equal to or greater than zero.

Field *max_args* indicates the maximum number of arguments the option requires. If the user specifies more arguments than this field indicates, an error message is produced. The value for this field must equal or exceed the value for min_args, and may not exceed UIF_MAX_ARGS.

Field *flags* is used to slightly modify the behavior of the command. Flag UIF_CMD_FLAG_HIDDEN prevents the option from being displayed in the SHOW menu.

Field *func* is the function to invoke when a command line matches the option name and meets its argument requirements. Function func receives two arguments. The first is the number of tokens on the command line, and the second is a pointer to an array of pointers pointing to each token on the command line. This scheme is similar to the invocation of the C language main() function.

Finally, field *syntax* describes the option usage and values. This information is displayed by SET.

Both the SET and SHOW commands use func. The indication of which command (SET or SHOW) invoked func is indicated in its first argument. If the value of the first argument is zero, one or two, then SHOW command invoked func to display option settings. If the value is zero, then the SHOW command is displaying all option values. When the value is three or greater, SET invoked func.

For examples, the common option entries are located in *dbug/src/include/dbug.h*.

7.3 Adding TFTP Download Support

The steps necessary for utilizing the TFTP Ethernet download are more complex, due to the Ethernet driver that must be written.

1. Edit *config.h* and #define DBUG_NETWORK.
2. Edit *<project>.c* and provide the board-specific functions needed during a network download. These functions are listed in Table 7-1 and are detailed in Section 10 Optional Board-Specific Functions.
3. Write the Ethernet driver. Details on writing an Ethernet driver are beyond the scope of this document; consult documentation for the Ethernet card or chip set. However, Ethernet drivers in *dbug/src/dev* provide examples on using the dBUG resources available to the driver.
4. The download functions in *<project>.c* need modification to accommodate the Ethernet download path (variable *uif_dlio* indicates the download type is UIF_DLIO_NETWORK). Function board_dlio_init() must register the interrupt handler and initialize the Ethernet driver. Function board_dlio_getchar() needs to call tftp_in_char(). Function board_dlio_done() must uninstall the interrupt handler().
5. Depending upon the interrupt scheme, the interrupt handler may need to explicitly clear the Ethernet interrupt. As such, the interrupt handler may be an intermediate function which clears the interrupt, and in turn invokes the real Ethernet driver interrupt handler.

FUNCTION	DESCRIPTION
board_dlio_filetype()	Determine download file type
board_irq_enable()	Enable Interrupts
board_irq_disable()	Disable Interrupts
board_set_client()	Set board IP address
board_get_client()	Get board IP address
board_set_server()	Set TFTP server IP address
board_get_server()	Get TFTP server IP address
board_set_gateway()	Set gateway IP address
board_get_gateway()	Get gateway IP address
board_set_netmask()	Set IP netmask
board_get_netmask()	Get IP netmask
board_set_filename()	Set default download filename
board_get_filename()	Get default download filename
board_set_filetype()	Set default download file type
board_get_filetype()	Get default download file type

Table 7-1 Optional Board-Specific Functions

To allow these changes to take effect, perform a *make<project>- clean* followed by a *make <project>*. (The definition of DBUG_NETWORK affects other conditional macros, thus requiring the *make clean* at least once.) Once built, the dBUG command DN performs the network download.

8. Resources Available to the Board Support Package

Many resources are available for use by board support packages. All of the following resources are defined in *dbug/src/include/dbug.h*.

8.1 Standard C Library

dBUG uses and provides several functions in the standard C library. By providing these standard C library functions, one dependency on the host toolchain is eliminated. Consult ANSI C for information on these functions.

- isspace(), isalnum(), isdigit(), isupper()
- strcmp(), strncmp(), strcasecmp(), strncasecmp()
- strtoul(), strlen(), strcat(), strncat(), strcpy(), strncpy()
- memcpy(), memset()
- printf(), sprintf()

NOTE:
printf() and sprintf() currently do not support floating
point formats, and %b indicates a binary format.

8.2 User Interface Resources

Certain routines in the User Interface are available for BSPs that implement new commands or obtain user input. Common messages are available as well.

- COPYRIGHT - dBUG copyright banner message.
- HELPMSG - Help banner.
- INVARG - Useful for error messages, contains “Error: Invalid argument: %s\n”.
- INVCMD - Contains “Error: Invalid command: %s\n”.
- INVREG - Contains “Error: Invalid register: %s\n”.
- INVALUE - Contains “Error: Invalid value: %s\n”.
- UIF_MAX_ARGS - This value is the maximum number of arguments allowed on the command line.
- UIF_MAX_LINE - This value is the maximum length of command line input.
- UIF_VER_MAJOR - This value indicates the major revision of the common user interface features.
- UIF_VER_MINOR - This value indicates the minor revision of the user interface.
- BASE - This variable indicates the user’s preference for converting strings to numbers.
- pause() - Function for providing rudimentary display paging.
- get_line() - Function for obtaining command line input.
- make_argv() - Function for parsing command line input into tokens.
- get_value() - Function to convert a string or symbol name into a 32-bit value.

Additional details for the functions are provided in Section 11 dBUG Internal Functions.

8.3 CPU Specific Resources

Certain routines and information in the CPU-specific portion are available for BSPs.

- CPU_STR - CPU name.
- CPU_VER_MAJOR - CPU-specific code major revision.
- CPU_VER_MINOR - CPU-specific code minor revision.
- context - This global data structure holds a copy of the CPU register context.

Many other functions exist which are used from within the User Interface, but are not available to board support packages.

8.4 Download Resources

The implementation for downloading files utilizes a simple byte-stream approach. During the download, board_dlio_getchar() is called to return the next byte in the data stream. However, the data stream can originate from a variety of sources. The source of the download data is set in the appropriate user command and the download process initiated. dBUG directly supports download via the console port, typically a serial port, or from an Ethernet network using TFTP.

- *uif_dlio* - this variable indicates the source of the download data stream, typically either UIF_DLIO_CONSOLE or UIF_DLIO_NETWORK.
- UIF_DLIO_CONSOLE - The data stream is obtained from the console port.
- UIF_DLIO_NETWORK - The data stream obtained via TFTP from the network.

When the download source is the network, dBUG processes the download data stream to accommodate ELF, COFF, S-Record and binary files. The file type (ELF, COFF, S-Record or binary) is indicated on the DN command line, or can be derived from the download filename extension.

- *.elf - Download file type is UIF_DLIO_ELF.
- *.coff - Download file type is UIF_DLIO_COFF.
- *.srec - Download file type is UIF_DLIO_SREC.
- *.bin - Download file type is UIF_DLIO_IMAGE.

Additional filename extensions can be associated with one of the above file types. The board-specific function board_dlio_filetype() returns one of the above file types, or UIF_DLIO_UNKNOWN. This function is documented in Section 10 Optional Board-Specific Functions.

As an example, these are the general steps for downloading from a parallel port.

1. Define the new download stream source, i.e. UIF_DLIO_PARALLEL in *<project>.h* (do not conflict with the sources defined in *dbug.h*).
2. Modify the functions board_dlio_init(), board_dlio_getchar() and board_dlio_done() to accommodate the new stream source.
3. Create the parallel port driver that is invoked from within board_dlio_init(), board_dlio_getchar() and board_dlio_done().
4. Create a new user command, i.e. DP, that sets *uif_dlio* to the value UIF_DLIO_PARALLEL, calls board_dlio_init(), then calls download_srecord() to perform an S-record download, and finishes by calling board_dlio_done().
5. Add the new command to the dBUG command set.

Once dBUG is rebuilt, the new DP command can be used to download S-records from the system's parallel port.

8.5 Interrupt Handling Resources

dBUG provides a method for hooking CPU interrupts. By registering an interrupt handler with dBUG, CPU register context save and restore operations are performed by dBUG, thus relieving the user of the need to manage context preservation.

- isr_register_handler() - This function installs an interrupt service routine.
- isr_remove_handler() - This function removes a previously installed interrupt service routine.
- ISR_DBUG_ISR - An argument to isr_register_handler(), this flag gives the handler priority over other handlers installed on the same interrupt vector with ISR_USER_ISR.
- ISR_USER_ISR - An argument to isr_register_handler(), this flag indicates a lower priority handler for the interrupt vector.

dBUG maintains a simple list of registered handlers. When an interrupt occurs, dBUG first examines the list for a match on the interrupt number and ISR_DBUG_ISR. If a match is found, the handler is invoked. If the handler returns FALSE, indicating that the interrupt was not serviced, the search continues for another match on the interrupt handler and ISR_DBUG_ISR. If the handler returns TRUE, then the search on ISR_DBUG_ISR stops. When dBUG completes its search for the interrupt number and ISR_DBUG_ISR, it then performs a search for the interrupt number and ISR_USER_ISR in the same fashion.

While dBUG itself is executing, all interrupts are disabled. For the CPU to recognize an interrupt and invoke any interrupt service routine, interrupts must explicitly be enabled. Furthermore, dBUG disables interrupts at all exception entry points, except handled interrupts. Note that default settings for CPU control registers enable interrupts when executing user code via the GO and GT commands.

Additional information on these functions is provided in Section 11 dBUG Internal Functions.

8.6 Miscellaneous Resources

The following are used by dBUG, and are available for general use.

- TRUE - Evaluates to 1.
- FALSE - Evaluates to 0.
- NULL - Evaluates to 0.
- __USER_SPACE - Address of memory available for general use (and not used by dBUG).

9. Board-Specific Functions

The following functions are needed for board support packages. These functions typically reside in the file *dbug/proj/<project>/src/<project>.c*. The *dbug/proj/common* project also provides some of these functions in their most common form.

board_init() Board Initialization Function

Syntax:

void board_init (void);

Description:

This is the first of three board initialization functions invoked by dBUG. This function performs the majority of board initialization.

Activities that must be completed by board_init() include: 1) dBUG console port initialization, and 2) other peripheral initialization.

Until the dBUG console port is initialized, printf() will not work.

When board_init() returns, dBUG performs internal initialization.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_init2()
board_init3()

board_init2() Board Initialization Function

Syntax:

    void board_init2 (void);

Description:

This function is the second of three board initialization functions. Any initialization of the board that draws on internal resources of dBUG may be performed here.

If not performed in board_init(), the console port used by dBUG must be initialized in this function. Upon returning from board_init2(), dBUG invokes printf() in displaying the start-up banner. Until the dBUG console port is initialized, printf() will not work.

At this point, initialization of dBUG is complete. If necessary, hooks can be placed in this function to perform operating system bootstrap or other system features.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_init()
board_init3()

board_init3() **Board Initialization Function**

Syntax:

void board_init3 (void);

Description:

This function is the third of three board initialization functions. Any initialization of the board that draws on internal resources of dBUG may be performed here.

Upon returning from board_init3(), dBUG displays the help message and the dBUG command prompt, dBUG>.

If necessary, hooks can be placed in this function to perform operating system bootstrap or other system features.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_init()
board_init2()

board_getchar() Character input

Syntax:

    char board_getchar (void);

Description:

This function obtains the character available on the dBUG console port.

This function must poll until a character is available.

Parameters:

None.

Return values:

Character input from dBUG console port.

Errors:

None.

See Also:

board_putchar()
board_getchar_present()

board_putchar()   Character output

Syntax:

```
void board_putchar (char ch);
```

Description:

This function outputs a character on the dBUG console port.

This function must not return until the character is output.

This function is called directly by printf().

Parameters:

ch	The character to output.

Return values:

None.

Errors:

None.

See Also:

board_getchar()
board_putchar_flush()

board_getchar_present() Test for character input

Syntax:

int board_getchar_present (void);

Description:

This function tests whether a character is available on the dBUG console port.

This function does NOT poll until a character is available, it merely tests for the presence of a character.

Parameters:

None.

Return values:

TRUE	Character is available.
FALSE	Character is not available.

Errors:

None.

See Also:

board_getchar()

board_putchar_flush() Flush character output

Syntax:

void board_putchar_flush (void);

Description:

This function is called prior to displaying the dBUG> prompt in order to flush output characters on the dBUG console port.

For dBUG console ports which are serial ports, this function is typically empty.

Parameters:

None.

Return values:

TRUE	Character is available.
FALSE	Character is not available.

Errors:

None.

See Also:

board_putchar()

board_dlio_getchar() Download character input

Syntax:

```
int board_dlio_getchar (void);
```

Description:

This function is called during a download to obtain the next data byte.

The global variable *uif_dlio* indicates the type of download being performed. When the DL command is invoked, *uif_dlio* indicates UIF_DLIO_CONSOLE. When the DN command is invoked, *uif_dlio* indicates UIF_DLIO_NETWORK.

For console downloads, this function returns the value obtained from board_getchar(). For network downloads, this function returns the value obtained from tftp_in_char().

Parameters:

None.

Return values:

Next character.

Errors:

None.

See Also:

board_dlio_done()
board_dlio_init()
board_dlio_vda()

board_dlio_init() Download initialization

Syntax:

int board_dlio_init (void);

Description:

This function is called prior to performing a download to perform initialization or activities needed to assist the download.

The global variable *uif_dlio* indicates the type of download being performed. When the DL command is invoked, *uif_dlio* indicates UIF_DLIO_CONSOLE. When the DN command is invoked, *uif_dlio* indicates UIF_DLIO_NETWORK.

For console downloads, typically there are no tasks for this function to perform. However, for network downloads, this function is required to perform two tasks: 1) register an interrupt service routine, and 2) initialize the network device.

For both console and network downloads, sometimes it is useful to enable instruction (but not data) caching at this time.

Parameters:

None.

Return values:

TRUE	Download can proceed.
FALSE	Download can not proceed.

Errors:

None.

See Also:

board_dlio_done()
board_dlio_getchar()
board_dlio_vda()

board_dlio_vda() Download valid address

Syntax:

int board_dlio_vda (ADDRESS addr);

Description:

This function is called during a download to determine if an address is a valid download address.

A given 32-bit address is not always a valid address for placing download data. An address that points to the dBUG reserved space, ROM, or I/O or un-populated RAM is an invalid address.

At a minimum, this function compares the provided address against the known dBUG reserved space and system RAM to determine if addr can be used to store download data.

Parameters:

addr	Address at which to download.

Return values:

TRUE	addr is valid address at which to download.
FALSE	addr is not a valid address at which to download.

Errors:

None.

See Also:

board_dlio_done()
board_dlio_init()
board_dlio_getchar()

board_dlio_done() Download completion

Syntax:

void board_dlio_done (void);

Description:

This function is called after completing a download to stop the download process.

The global variable *uif_dlio* indicates the type of download performed. When the DL command is invoked, *uif_dlio* indicates UIF_DLIO_CONSOLE. When the DN command is invoked, *uif_dlio* indicates UIF_DLIO_NETWORK.

For console downloads, typically there are no tasks for this function to perform. However, for network downloads, this function is required to de-register the interrupt service routine and graceful turn off the Ethernet device.

If instruction caching is enabled during the download, then this function should disable caching.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_dlio_init()
board_dlio_getchar()
board_dlio_vda()

board_get_baud() Get baud rate of dBUG port

Syntax:

```
int board_get_baud (void);
```

Description:

This function is called to obtain the baud rate of the dBUG console port. The value returned by this function is used in configuring the console port at boot time.

This function is also invoked by the SHOW BAUD command.

To fix the baud rate at a particular value, simply return the desired value and make board_set_baud() do nothing.

If possible, the baud rate value should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

None.

Return values:

Typical values are 9600, 19200 and 38400.

Errors:

None.

See Also:

board_set_baud()

board_set_baud() Set baud rate of dBUG port

Syntax:

```
void board_set_baud (int baud);
```

Description:

This function is called to set the baud rate of the dBUG console port. Because this value is used in configuring the console port at boot time, it is helpful if this value is stored in persistent memory, i.e. non-volatile RAM.

This function is invoked by the SET BAUD command.

If a fixed baud rate is being used, then this function should be empty.

If possible, the baud rate value should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

baud	Typical values are 9600, 19200 and 38400.

Return values:

None.

Errors:

None.

See Also:

board_get_baud()

board_reset() Board reset

Syntax:

```
void board_reset (void);
```

Description:

This function is invoked by the RESET command to reset the board.

This function contains code for a software-initiated reset. If no such mechanism exists, then this function is empty and dBUG executes the same code sequence as if a hard reset occurred.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

None.

10. Optional Board-Specific Functions

These functions are needed only if the TFTP download feature is utilized.

board_dlio_filetype() — Determine download file type

Syntax:

```
int board_dlio_filetype (char *fn, char *ext);
```

Description:

This function determines the download file type (ELF, COFF, S-Record or Binary) from the filename. dBUG examines the extension (that part of the filename following the period, if one exists) to determine the download file type if none is specified on the DN command line.

Parameters:

fn	Pointer to the string containing the download filename.
ext	Pointer to the filename extension

Return values:

UIF_DLIO_UNKNOWN	Download file type is not known.
UIF_DLIO_ELF	Download file type is ELF.
UIF_DLIO_COFF	Download file type is COFF.
UIF_DLIO_SREC	Download file type is S-Record.
UIF_DLIO_IMAGE	Download file type is binary data.

Errors:

None.

See Also:

None.

board_irq_enable() Enable board interrupts

Syntax:

void board_irq_enable (void);

Description:

This function is used to enable board interrupts during the network download. It is used in conjunction with board_irq_disable() to delineate critical section processing during the download.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_irq_disable()

board_irq_disable() Disable board interrupts

Syntax:

```
void board_irq_disable (void);
```

Description:

This function is used to disable board interrupts during the network download. It is used in conjunction with board_irq_enable() to delineate critical section processing during the download.

Parameters:

None.

Return values:

None.

Errors:

None.

See Also:

board_irq_enable()

board_set_client() Set board IP address

Syntax:

```
void board_set_client (uint8 *ipaddr);
```

Description:

This function is used to store the Internet Protocol (IP) address of the board in persistent storage. This function is invoked when the user enters the SET CLIENT command.

If possible, the client IP should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to the 4-byte IP address.

Return values:

None.

Errors:

None.

See Also:

board_get_client()

board_get_client() Get board IP address

Syntax:

```
uint8 * board_get_client (uint8 *ipaddr);
```

Description:

This function is used to retrieve the IP address of the board from persistent storage. This function is invoked when the user enters the SHOW CLIENT command, and by the DN command.

If possible, the client IP should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to buffer for copying the 4-byte IP address.

Return values:

Pointer to the 4-byte IP address.

Errors:

None.

See Also:

board_set_client()

board_set_server() Set server IP address

Syntax:

```
void board_set_server (uint8 *ipaddr);
```

Description:

This function is used to store the IP address of the server in persistent storage. This function is invoked when the user enters the SET SERVER command.

If possible, the server IP should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to the 4-byte IP address.

Return values:

None.

Errors:

None.

See Also:

board_get_server()

board_get_server() Get server IP address

Syntax:

```
uint8 * board_get_server (uint8 *ipaddr);
```

Description:

This function is used to retrieve the IP address of the server from persistent storage. This function is invoked when the user enters the SHOW SERVER command, and by the DN command.

If possible, the server IP should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to buffer for copying the 4-byte IP address.

Return values:

Pointer to the 4-byte IP address.

Errors:

None.

See Also:

board_set_server()

board_set_gateway() — Set gateway IP address

Syntax:

```
void board_set_gateway (uint8 *ipaddr);
```

Description:

This function is used to store the IP address of the gateway in persistent storage. This function is invoked when the user enters the SET GATEWAY command.

If possible, the gateway IP should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to the 4-byte IP address.

Return values:

None.

Errors:

None.

See Also:

board_get_gateway()

board_get_gateway() Get gateway IP address

Syntax:

```
uint8 * board_get_gateway (uint8 *ipaddr);
```

Description:

This function is used to retrieve the IP address of the gateway from persistent storage. This function is invoked when the user enters the SHOW GATEWAY command, and by the DN command.

If possible, the gateway IP should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipaddr	Pointer to buffer for copying the 4-byte IP address.

Return values:

Pointer to the 4-byte IP address.

Errors:

None.

See Also:

board_set_gateway()

board_set_netmask() Set IP netmask

Syntax:

```
void board_set_netmask (uint8 *ipmask);
```

Description:

This function is used to store the IP netmask in persistent storage. This function is invoked when the user enters the SET NETMASK command.

If possible, the IP netmask should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

ipmask	Pointer to the 4-byte IP netmask.

Return values:

None.

Errors:

None.

See Also:

board_get_netmask()

board_get_netmask() Get IP netmask

Syntax:

```
uint8 * board_get_netmask (uint8 *ipmask);
```

Description:

This function is used to retrieve the IP netmask from persistent storage. This function is invoked when the user enters the SHOW NETMASK command, and by the DN command.

If possible, the IP netmask should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

ipmask	Pointer to buffer for copying the 4-byte IP netmask.

Return values:

Pointer to the 4-byte IP netmask.

Errors:

None.

See Also:

board_set_netmask()

board_set_filename() Set default download filename

Syntax:

void board_set_filename (char *filename);

Description:

This function is used to store the default filename for network downloads in persistent storage. This function is invoked when the user enters the SET FILENAME command.

If possible, the filename should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

filename	Pointer to the filename.

Return values:

None.

Errors:

None.

See Also:

board_get_filename()

board_get_filename() Get default download filename

Syntax:

```
char * board_get_filename (char *filename);
```

Description:

This function is used to retrieve the default network download filename from persistent storage. This function is invoked when the user enters the SHOW FILENAME command, and by the DN command.

If possible, the filename should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

filename	Pointer to buffer for copying the filename.

Return values:

Pointer to the filename.

Errors:

None.

See Also:

board_set_filename()

board_set_filetype() Set default download file type

Syntax:

void board_set_filetype (int filetype);

Description:

This function is used to set the default file type for network downloads. This function is invoked when the user enters the SET FILETYPE command.

If possible, the file type should be stored in persistent storage, i.e. non-volatile RAM.

Parameters:

filetype	The download file type, either UIF_DLIO_SREC, UIF_DLIO_ELF, UIF_DLIO_COFF or UIF_DLIO_IMAGE.

Return values:

None.

Errors:

None.

See Also:

board_get_filetype()

board_get_filetype() Get default download file type

Syntax:

```
int board_get_filetype (void);
```

Description:

This function is used to retrieve the default network download file type. This function is invoked when the user enters the SHOW FILETYPE command, and by the DN command.
If possible, the file type should be obtained from persistent storage, i.e. non-volatile RAM.

Parameters:

None.

Return values:

The download file type, either UIF_DLIO_ELF, UIF_DLIO_COFF, UIF_DLIO_SREC or UIF_DLIO_IMAGE.

Errors:

None.

See Also:

board_set_filetype()

11. dBUG Internal Functions

These functions are provided in dBUG and are usable by the board support package.

pause() Simple console pagination

Syntax:

int pause (int *rows);

Description:

This function provides a simple method for console pagination. On each invocation of this function, the value rows is incremented by one. When the value of rows surpasses 21, the banner is displayed and awaits user input to continue.

```
Press <ENTER> to continue.
```

This function is NOT invoked automatically by printf(); instead, this function must be explicitly invoked when pagination is desired. To use this function, the user must initialize a local integer variable to the value zero, then after displaying a single line of output, invoke pause(). The variable rows should track the number of lines of output displayed to the console. When the user selects <Enter> to continue, the variable rows is reset to zero.

The user may select <Enter> to continue, and q or Q or <Ctrl> C to indicate a desire to abort the display.

Parameters:

rows	Integer value indicates current number of lines of output displaying on the console.

Return values:

TRUE	User selected q or Q or <Ctrl> C.
FALSE	User selected <Enter>.

Errors:

None.

See Also:

None.

get_line() Console input

Syntax:

```
char * get_line (char *line);
```

Description:

This function obtains a line of input from the dBUG console and places it into the user-supplied character buffer line. The backspace and delete keys provide a rub-out feature, the only editing capability. This function returns when the user has pressed the <Enter> key. The buffer is properly terminated.

The character buffer must be of size UIF_MAX_LINE or greater.

Parameters:

line	Pointer to the character buffer for the user input.

Return values:

This function returns a pointer to the head of the character buffer. This value is equivalent to the value of line.

Errors:

None.

See Also:

make_argv()

make_argv() Parse string into tokens

Syntax:

```
int make_argv (char *line, char *argv[]);
```

Description:

This function parses the NULL-terminated string line into tokens, and places pointers to the resulting tokens into argv[]. This function is commonly used to process user input.

Tokens are delineated by white space. As the function scans the string, it places the NULL character \0 into the string in place of white space. In doing so, the token becomes a properly NULL-terminated string. The number of tokens parsed is returned.

The token list argv[] must be a minimum size UIF_MAX_ARGS plus one for a NULL terminated list. The token list is NULL terminated upon completing the scan.

This function modifies the original string.

Parameters:

line	Pointer to the character buffer to be processed. This buffer will be modified.
argv	Pointer to the array of character pointers which point to the tokens. This array is NULL terminated.

Return values:

This function returns the number of tokens parsed.

Errors:

None.

See Also:

get_line()

get_value() Convert string to number

Syntax:

```
uint32 get_value (char *str, int *success, int base);
```

Description:

This function converts the string str into a 32-bit unsigned number.
The function first attempts to locate the string in the symbol table. If a match is found, the value of the symbol is returned.
Otherwise the function converts the string according to radix base. The radix is a value 2 for binary, 8 for octal, 10 for decimal or 16 for hexadecimal. The value 0 for radix indicates that get_value() should determine the radix by examining the string for radix indicators.

Parameters:

str	Pointer to the character string to be converted.
success	Pointer to an integer. This variable indicates whether or not the conversion encountered errors.
base	The radix for converting the string. This value must be between 0 and 16 (inclusive).

Return values:

If no errors are encountered, this function returns a 32-bit unsigned value and success is TRUE. If errors are detected, the value zero is returned, and success is FALSE.

Errors:

For a given radix, certain characters are valid. Hexadecimal notation, for example, allows the letters ‘0’ through ‘9’ and ‘A’ through ‘F’, but not the letter ‘G’. If an illegal character is encountered, then success contains FALSE.

See Also:

strtoul()

isr_register_handler() Install an Interrupt Service Routine

Syntax:

```
int isr_register_handler (int type, int vector,int (*handler)(void *arg1, void *arg2),
                          void *arg1, void *arg2);
```

Description:

This function installs an interrupt service routine (ISR) for the indicated vector. The type provides a relative priority for handlers which may be installed on the same vector: type ISR_DBUG_ISR is serviced prior to ISR_USER_ISR. This scheme allows dBUG to prioritize internal interrupt handlers over user-installed interrupt handlers.

When an interrupt occurs, dBUG saves the registers on the stack and searches the list of registered interrupt service routines. If an ISR for the appropriate vector is located, dBUG executes the ISR by invoking

```
handler(arg1, arg2);
```

dBUG examines the return value of the ISR to determine whether the interrupt was successfully serviced. If the return value is TRUE, then dBUG restores the registers and continues execution. Otherwise, dBUG dumps the register set to the console and displays the dBUG> prompt.

NOTE: While any vector can be passed to this routine, the ISR will only be invoked for vectors that are actually CPU interrupt vectors. For example, installing an interrupt handler for the same vector as the bus exception vector will never be invoked because the exception handling for the bus exception never examines the list of interrupt service routines.

Parameters:

type	Relative priority type of interrupt: ISR_DBUG_ISR or ISR_USER_ISR.
vector	CPU-specific vector number for the interrupt.
handler	The address of the interrupt service routine.
arg1	Pointer to an implementation-specific value or data structure.
arg2	Pointer to an implementation-specific value or data structure.

Return values:

If TRUE is returned, the handler was successfully installed. Otherwise, the handler was not installed.

Errors:

None.

See Also:

isr_remove_handler()

isr_remove_handler() Remove an Interrupt Service Routine

Syntax:

```
int isr_remove_handler (int (*handler)(void *arg1, void *arg2));
```

Description:

This function removes an interrupt handler for handler previously installed with isr_register_handler().

Parameters:

handlerInterrupt service routine address.

Return values:

If TRUE is returned, the handler was successfully un-installed.

Errors:

None.

See Also:

isr_register_handler()