Debugging with C and ELF/DWARF

The In-Circuit Debugger supports loading files of type ELF/DWARF version 2.0, providing C source-level debugging. The ICD will load files that adhere to the following specifications:

- Executable and Linking Format (ELF)
- System V Application Binary Interface, edition 4.1
- Debug With Arbitrary Record Format (DWARF)
- DWARF Debugging Information Format Revision 2.0.0

Note that the debugger will load DWARF version 2.0 information only. No other DWARF version is compatible with the ICD. Some processors have ELF/DWARF supplemental specifications in addition to the general specifications above. The ICD adheres to all available processor supplements.

1. SUPPORTED FEATURES

- Source-Level Code View
- Disassembly-Level Code View
- Single-Stepping Source
- Running and Stopping Source
- Loading Object Data to MCU Memory:
- Base Type Variable View
- Array Variable View
- Structure/Union Variable View
- Enumerated Type Variable View
- Typedef Variable View
- Global Variable View
- Location Relative (e.g. Stack Relative) Variable View
- Register Based Variable View
- Variable Scoping:
- Auto-typing of Variables in VAR Window
- Modifying Variables
- Currently, the ICD does not support C macro information.

2. C DEBUGGING COMMANDS

Use the following commands to debug with ELF/DWARF:

- The HLOAD command loads ELF/DWARF debug information and loads the object code to target RAM.
- The HLOADMAP command loads DWARF debugging information only, ignoring target object code.
- The HSTEP command, or SS, single-steps one high-level source line.
- The HSTEPFOR command continually steps high-level instructions until the user aborts by pressing a key.
- The HGO command starts full-speed execution and, when aborted by the user, attempts to stop on a high-level instruction.
- The HSTEP, HSTEPFOR, and HGO commands are identical to the STEP, STEPFOR, and GO commands with the **following exceptions**:

While assembly instructions are executed between the high-level language lines, the Variable, Memory, and Source Code windows are not updated. The Disassembly and CPU windows are updated for every low-level instruction. At the next high-level instruction boundary, all windows are updated (unless in GO mode).

When the user aborts the current execution command, the debugger executes up to 20 steps while trying to find the next high-level language instruction boundary. The debugger will attempt to show source automatically. If, in 20 steps, it cannot find the boundary of a high-level source code instruction, it will stop and show disassembly. To see source again, use the HSTEP command or right click on the Source Code window and select Show Source Module.

3. LOADING THE ELF/DWARF 2.0 FILE

The ELF/DWARF 2.0 file contains two types of information:

a.) ELF Binary Image: This includes all instructions and data that comprise the target application. This image may reside in RAM or flash.

b.) DWARF Debug Information: This contains source file and variable information. The debugger uses this information to allow the user to debug the binary object code. The debug information resides in the debugger and not on the target microcontroller.

There are two common situations when loading an ELF/DWARF 2.0 file:

a.) Binary image programmed to flash: The ICD does not program flash. Program flash prior to entering ICD using P&E's PROG flash programmer. Upon entering the ICD, with the binary image already resident in flash, use the HLOADMAP command to load the debug portion of the ELF/DWARF file.

b.) Binary image programmed to RAM: Use the HLOAD command to load both the binary image to the target RAM and the debug information to the debugger. Before loading the binary image, the user should make sure that the RAM is enabled at the proper target address. After using HLOAD or HLOADMAP, the debugger status window will display the amount of debug and object information loaded from the ELF/DWARF file. By default, the debugger sets the program counter (PC) to the initial PC value of application code.

4. EXECUTING TO THE BEGINNING OF SOURCE CODE

After loading an ELF/DWARF file, the code window may point to disassembly and not show any of the user's source code. ICD must be able to locate the source files that is listed in the ELF/DWARF file. The path information may be either relative or absolute. While in ICD, the user can right click on the Code Window and then Select Source Module to see the path information. All relative paths are interpreted to the location of the ELF/DWARF file. The user may move the source files so that the relative paths make sense or try to configure the compiler to use absolute paths.

Before running the user's main () function, the compiler must first execute initialization code that may not have corresponding debug information. To run past the compiler initialization code, issue the "gotil main" command in the status window. Note that the labels are case sensitive and that the main label should be lowercase. This will set a breakpoint at the beginning of the main() routine and start the processor running. Execution should stop almost immediately, and the PC should be pointing to valid source code.

If the main() symbol is not in scope initially, manually set a breakpoint at the beginning of main() and issue the GO command.

Alternatively, instead of using the GOTIL command, the user could step through the initialization disassembly code using the STEP or HSTEP commands, eventually reaching the main() function.

If there is no initialization code, the SHOWCODE main command may be used.

5. STEPPING THROUGH C SOURCE CODE

The P&E debugger implements a high-level language source step command. Use the HSTEP command in the status window or click the high-level step button on the debugger button bar. At every HSTEP, the debugger will rapidly single step assembly level instructions until the next source instruction, and execution will cease. While the debugger is fast single-stepping, the debugger updates the PC value only. When the debugger reaches the next source instruction, it will update all visible windows with data from the target. Note that using the HSTEP command does not run code in real-time. Real-time execution is described in the next section.

Note that some instructions will take longer to step than others, because each C level instruction may consist of a greater or fewer number of underlying assembly instructions.

6. DISPLAYING VARIABLES

The debugger Variable window will show global and static variables as well as location relative variables. A location relative variable is typically a local variables on the application stack. However, the compiler may indicate other variable types as location relative and may use many different location schemes for variables. Some variables may change location depending on the value of the PC. The ICD supports all of the DWARF 2.0 location posibilities.

The debugger will show variables whose location is a register. Compilers will often store temporary variables in CPU registers of the processor. If you attempt to look at the address of a register variable by adding the symbol &varName to the Variables window (where "varName" is the variable name), the debugger will indicate the register in which the variable is stored.

The ICD supports scoping of variables. If you enter a variable name in the Variables window, the debugger will show the variable of the same name that is currently in scope. If you had a global integer variable, temp, and a local float variable, temp, within the routine init_port, the float would be shown while you step through the init_port routine. Otherwise, the Variable window would display the integer variable.

The following symbols may be added to a variable name in the Variables window. Note that pointer variables are displayed in red.

& dereference

* reference

. access to union or structure member

-> pointer access to union or structure member

[ ] array subscript

For example:

```
int TintGlobal;

int *ptrTint;

int TmultiArray[3][3][3];

struct Tstruct {
```

```
  int ii;
  int jj;
  short ll;
} TstructInstance;

union Tunion {
  unsigned long TuLongUnion;
  struct Tstruct TstructUnion
} TunionInstance;

union Tunion *TunionPointer;
```

ICD commands:

var TintGlobal

The value of TintGlobal

var &TintGlobal

The address of TintGlobal

var *ptrTint

The value of the variable pointed to by ptrTint

var TmultiArray[0][1][2]

The value of this element of the array

var TstructInstance.ii

The value of this member of the structure

var TunionPointer->TuLongUnion

The value of this union member, the union pointed to by TunionPointer

7. ELF PROGRAM HEADERS

Program headers, included in every executable ELF/DWARF file, describe how the application object code is to be loaded to the target. Two values in each program header entry, defined by the System V ABI as p_paddr and p_vaddr, are available to provide a load address for a particular group of code. For executable files, the p_vaddr field is typically used to provide the load address. However, some compilers, such as the GNU compiler, may utilize the p_paddr field instead.

When the P&E debugger loads the ELF/DWARF file, it may detect the use of the non-standard p_paddr field in the ELF program header. In this instance, the debugger will display a dialog box that will ask the user what to do. Generally, when using the GNU tools, click "Yes" in the dialog box to load code using the non-standard p_paddr field.

For a complete description of ELF Program Headers, see the System V ABI (ELF) specification.