

Debugging and Diagnostic Tools

This chapter describes High C/C++ debugging and diagnostic aids. It contains the following sections:

- §13.1: *Debugger Support*
- §13.2: *Tracing Function Calls*
- §13.3: *Minimizing Symbolic Debug Information*

13.1 Debugger Support

The `-g` option directs the compiler to generate debugger information. We do not recommend using option `-g` with the `-On` option, because the optimizations can confuse the debugger. Most level 0 (zero) optimizations are performed anyway.

However, when you specify `-g`, stores are not delayed and dead stores are not removed. Debuggers typically function adequately at this level. The only exception is that variables cannot be reliably modified from the debugger.

Table 13.1 *Debugger Support for PowerPC Applications*

Target Platform	Debug Info Format
All platforms on PowerPC except Solaris	DWARF 1+
Solaris on PowerPC	Sun <code>dbx</code> <code>stabs</code>

13.2 Tracing Function Calls

The High C/C++ compiler can generate code to trace entry to, exit from, and calls to functions in your application. Each time one of these events occurs, the appropriate trace function is called.

Simple versions of these trace functions are included in the library source directory, in file `trace.c`. You can customize the functions to meet your particular needs. Within a trace function you can, for example, print the name of the function entered, exited, or called, or observe the function-call behavior on a particular set of inputs. Before you can use the functions, you must compile and include them.

One use of the tracing facility is to help you understand code you have not written yourself.

To enable function-call tracing, compile your code with the following toggles turned On:

Toggle	Tracing Function
Prolog_trace	Calls function <code>_prolog_trace()</code> when execution enters a function
Epilog_trace	Calls function <code>_epilog_trace()</code> when execution exits a function
Call_trace	Calls function <code>_call_trace()</code> just before a function is called

The tracing functions have the following signatures:

```
_prolog_trace(const char *func);
_epilog_trace(const char *func);
_call_trace(  const char *from_proc,
              const char *from_module,
              unsigned   line,
              const char *to_proc);
```

The parameters of these functions represent the following values:

Function	Parameter	Value
<code>_prolog_trace()</code>	<code>func</code>	Name of the function entered
<code>_epilog_trace()</code>	<code>func</code>	Name of the function exited
<code>_call_trace()</code>	<code>from_proc</code>	Name of the function making the call
	<code>from_module</code>	Source module from which the call is made
	<code>line</code>	Line of the source module where the call is made
	<code>to_proc</code>	Name of the function called

13.3 Minimizing Symbolic Debug Information

Debugging at the source-code level requires that the debugger be supplied with *symbolic debug information* (SDI). Traditionally, compilers have generated that information for each compilation unit compiled with the debug option (`-g`).

If two distinct compilation units compiled with option `-g` include the same header file (for example, `x.h`), SDI for the declarations within `x.h` is placed in the object module generated for each of the two compilation units. This means redundant SDI takes up space in both object modules and in the linked program.

If the program does not involve C++ code and the `Dwarf` toggle is `Off`, you can use linker option `-s` to direct the linker to eliminate duplicate `struct`, `union`, and `enum` definitions.

However, if the program does involve C++ code, DWARF information is required and the size of SDI increases dramatically. This is especially true because C++ `class` declarations require much more SDI than do C `struct` declarations.

MetaWare's solution to this problem is to cause the compiler to generate exactly *one* copy of SDI under DWARF for a C or C++ `struct` or `class`, no matter how many times that `struct` or `class` is referenced. This keeps

the amount of generated SDI to a minimum (and so speeds linking) and reduces the size of linked programs.

SDI minimization applies only to structures (including classes and unions), because they are the source of most of the redundant SDI for C and C++. The structure *must* be named; for example, **struct** Name, not **struct**{...}.

13.3.1 How the Compiler Minimizes SDI

SDI minimization is implemented with the following toggles, which cause the compiler to generate only one copy of SDI for a given **struct**:

- Autodebug
- Forcedebug
- Nodebug

Debug repository The simplest way to enable SDI minimization is to turn On toggle Autodebug for all your compilations. Turning On this toggle causes the compiler to place SDI for the named **struct** in the module where the **struct**'s first non-pure-virtual, non-inline function is defined. This module is called the *debug repository* of the **struct**.

Toggles Nodebug and Forcedebug provide more precise control over SDI generation. They minimize SDI for **structs** that have no non-pure-virtual, non-inline functions. With these toggles you can prevent the compiler from generating SDI for any named **struct** until you specifically instruct it to do so. In this way you can manually determine the **struct**'s debug repository.

See Chapter 4: *Using Compiler Toggles* for more information.

13.3.1.1 Enable SDI Minimization: Toggle Autodebug

To place SDI for the named **struct** in the debug repository of the **struct**, turn On toggle Autodebug. The compiler gives the SDI a name related to the name of the **struct** (which is why the **struct** must be named). All other modules that require the SDI reference the named SDI. References to the named SDI are resolved by the linker, just as the linker normally resolves references to an **extern** variable to its definition. For example:

File `st.h`:

```
struct s {
    s() {}
};
struct t {
    t();
};
```

File `t.cpp`:

```
#include "st.h"
t::t() {}
```

File `use.cpp`:

```
#include "st.h"
t x;
```

Compiling this source with `-g` and `-Hnon=Autodebug` causes named SDI for **struct** `t` to appear in `t.cpp`'s object module. Compiling `use.cpp` with the same switches causes a *reference* to `t`'s SDI to be generated; the compiler does *not* generate SDI for `t` when it compiles `use.cpp`.

Because **struct** `s` has no debug repository — its only function is **inline** — the compiler generates SDI for **struct** `s` whenever it is needed; this may result in redundant SDI for `s`.

13.3.1.2 Disable SDI Generation: Toggle Nodebug

Toggle Nodebug tells the compiler not to generate SDI for named **structs** declared within its scope. Turn it On in your header files:

File `st.h`:

```
#pragma On(Nodebug)
struct s {
    s() {}
};
struct t {
    t();
};
#pragma Pop(Nodebug)
```

Now, no matter where `s` and `t` are used in your program, the compiler does *not* generate SDI for them, only references to their SDI. However, at some

point you must generate SDI for `s` and `t`. You force this to happen with `toggle Forcedebug`.

13.3.1.3 Force Generation of SDI: Toggle Forcedebug

`Toggle Forcedebug` causes the compiler to generate SDI for any named **structs** that are declared in the scope of both `toggle Nodebug` and `toggle Forcedebug`. For example:

File `stdebug.cpp`:

```
#pragma On(Forcedebug)
#include "st.h"
#pragma Pop(Forcedebug)
```

Compiling `stdebug.cpp` generates SDI for both `s` and `t` and puts it in `stdebug`'s object module. In this way you can, for example, create a single debugging object module in your program that contains the debug information for all your header files.

You can change your code, compile and link your program with option `-g`, and include the debugging object module in your link to provide access to the SDI. You must recreate the debugging object file only when you change any of your named **structs**, so that you need updated SDI. For example, suppose you have the following header files:

File `h1.h`:

```
#pragma On(Nodebug)
struct one { ... };
...
#pragma Pop(Nodebug)
```

File `h2.h`:

```
#pragma On(Nodebug)
struct two { ... };
...
#pragma Pop(Nodebug)
```

File `h3.h`:

```
#pragma On(Nodebug)
struct three { ... };
...
#pragma Pop(Nodebug)
```

Given these header files, you must recompile your debug module, `debug.cpp`, only when any of the header files change:

File `debug.cpp`:

```
#pragma On(Forceddebug)
#include "h1.h"
#include "h2.h"
#include "h3.h"
#pragma Pop(Forceddebug)
```

