cādence®

Xtensa® Software Development Toolkit

User's Guide

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date:10/2018

RI-2018.0

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Figures

List of Tables

Preface

This document is written for customers who are experienced in the programming and debugging of software.

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- $ represents your shell prompt, in user-session examples.
- **`literal_input`** indicates literal command-line input.
- *`variable`* indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *`... output ...`* indicates unspecified program output.
- *`[optional-variable]`* indicates an optional parameter.
- **`[`***`variable`***`]`** indicates a parameter within literal square-braces.
- **`{`***`variable`***`}`** indicates a parameter within literal curly-braces.
- **`(`***`variable`***`)`** indicates a parameter within literal parentheses.
- | means *OR*.
- *`(var1 | var2)`* indicates a required choice between one of multiple parameters.
- *`[var1 | var2]`* indicates an optional choice between one of multiple parameters.
- *`var1 [, varn]*`* indicates a list of 1 or more parameters (0 or more repetitions).
- `4'b0010` is a 4-bit value specified in binary.
- `12'o7016` is a 12-bit value specified in octal.
- `10'd4839` is a 10-bit value specified in decimal.
- `32'hff2a` or `32'HFF2A` is a 32-bit value specified in hexadecimal.

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *flush* is deprecated due to potential ambiguity (it may mean *write-back* or *discard*).
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

Changes from the Previous Version

1. Overview

The Xtensa architecture is Cadence's configurable and extensible processor architecture. Specific processor configurations are created using the Xtensa Xplorer. For each processor configuration, we provide a tailored set of software development tools.

This guide provides an overview of the software development tools and shows a typical flow and methodology for using the tools. This document contains detailed information about using the software development tools in the command-line environment and an introduction to using Xtensa Tools with the Xtensa Xplorer Integrated Development Environment (IDE).

For configurable Xtensa cores, the software tools are delivered in two separate packages:

- Configuration-independent Xtensa Tools: This package includes all the Xtensa software development tools, including compilers, debuggers, instruction set simulator, and TIE compiler. These tools are identical across different Xtensa processor configurations and need to be installed only once. Configuration-independent tools need information and files from configuration-specific packages to work correctly.
- Configuration-specific core files: These files must be downloaded and installed for each build of a configuration. Each Xtensa configuration has a unique name, enabling you to develop code simultaneously on more than one processor configuration.

Some configuration-specific core files are already bundled together with the Xtensa Tools package.

To install the tools on your system, follow the directions in the *Xtensa Development Tools Installation Guide*.

1.1 Software Development Tools

Table 1 lists the GNU tool suite and other tools that have been tailored to your processor configuration that the Software Development Toolkit includes.

Table 1. Software Development Toolkit Tools

Tool	Description
`xt-cov`	Emit source code coverage information
`xt-xcc`	Xtensa C Compiler
`xt-xc++`	Xtensa C++ Compiler
`xt-clang`	LLVM-based Xtensa C Compiler

Table 1. Software Development Toolkit Tools (continued)

Tool	Description
`xt-clang++`	LLVM-based Xtensa C++ Compiler
`xt-as`	GNU Assembler
`xt-ld`	GNU Linker
`xt-run`	Xtensa Instruction Set Simulator (ISS)
`xt-gdb`	GNU Debugger
`xt-genldscripts`	Create linker scripts
`xt-genvpmodel`	Generate a wrapper to connect ISS to Synopsys Virtual Prototyping Environment
`xt-gprof`	GNU Profiler
`xt-regenlsps`	Copy a set of linker scriptsand runs xt-genldscripts on each.
`xtsc-run`	XTSC-based multi-core subsystem simulator
`xt-trace`	Converts trace port signals to assembly instructions
`xt-traxcmd`	Controls TRAX trace capture
`xt-traxview`	Converts TRAX compressed trace output to instruction execution history
`xt-link-order`	Profile-guided section reordering

1.2 Software Development Binary Utilities

In addition to the tools listed above, the software development toolkit also includes a set of binary utilities for use with the above tools. Table 2 lists these utilities.

Table 2. Software Development Toolkit Binary Utilities

Tool	Description
`xt-addr2line`	Converts addresses into file names and line numbers
`xt-ar`	Creates, modifies and extracts from archives
`xt-config`	Return information about configuration used to build object or binary.
`xt-c++filt`	Demangles C++ symbols
`xt-dumpelf`	Convert an elf binary into Verilog $readmh format or XTSC memory format
`xt-nm`	Lists symbols from object files
`xt-objcopy`	Copies and translates object files
`xt-objdump`	Displays information from object files
`xt-pkg-loadlib`	Package a loadable library into an object file.
`xt-ranlib`	Generates index to archive
`xt-readelf`	Displays information about ELF format object files

Table 2. Software Development Toolkit Binary Utilities (continued)

Tool	Description
`xt-size`	Lists section sizes and total size
`xt-strings`	Prints the strings of printable characters in files
`xt-strip`	Discards symbols from object files

1.3 Overview of Tool Generation

Figure 1 shows an overview of how the software development and hardware development tools for Xtensa are generated and used. The left-most column, below the Configuration-Specific Database, shows the steps relating to the software development tools described in this guide.

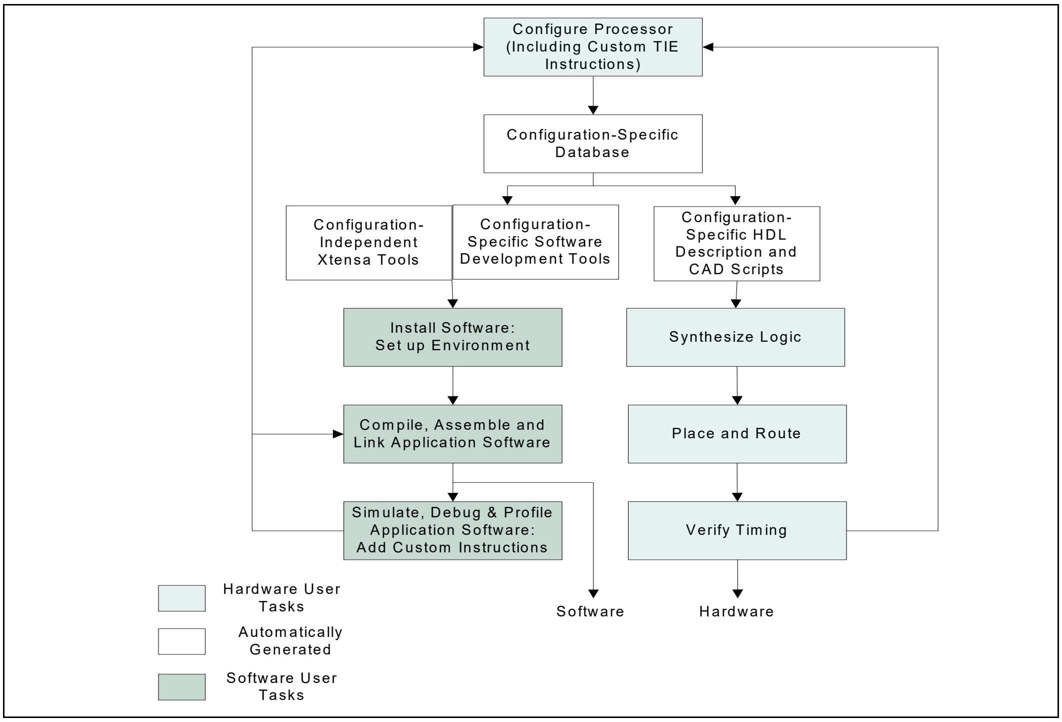

Configure Processor (Including Custom TIE Instructions)
Configuration-Specific Database
Configuration-Independent Xtensa Tools
Configuration-Specific Software Development Tools
Configuration-Specific HDL Description and CAD Scripts
Install Software: Set up Environment
Synthesize Logic
Compile, Assemble and Link Application Software
Place and Route
Simulate, Debug & Profile Application Software: Add Custom Instructions
Verify Timing
Software
Hardware
Hardware User Tasks
Automatically Generated
Software User Tasks

Figure 1. Overview of the Xtensa System

1.4 Supported Targets

Software compiled with the Xtensa tools will run on the following targets:

- *Xtensa Instruction Set Simulator (ISS):* This `xt–run` program is tailored to your specific processor configuration. The XTMP and XTSC libraries allow you to build system simulators for a uni- or multi-processor subsystem. The `xtmp-run` and `xtsc-run` allow you to simulate simple multiprocessor systems without having to create custom models in C or C++.
- Xtensa processor-based chips and emulation boards.

1.5 Supported Hosts

The software development tools run on Red Hat Enterprise 5 and 6 Linux and Microsoft Windows 7, 8, and 10 operating systems.

Note: The Xtensa software tools do not support spaces in any path names (for example, do not install tools in the `Program Files` folder on Windows).

1.6 Object File Formats

The software development tools support the Extended Linking Format (ELF) object-file format defined by the UNIX System V Application Binary Interface.

1.7 Case Sensitivity

Case sensitivity varies among the software development tools, as follows:

- *Case-Sensitive Strings:*
  - Command-line options
  - Assembler labels
  - Linker-script commands
  - C and C++ intrinsics
- *Case-Insensitive Strings:*
  - Debugger commands
  - Assembler instructions and register names
  - File and path names on Windows

2. Xtensa Xplorer and Xtensa Tools

Xtensa Xplorer integrates software development, processor optimization and multiple-processor SOC architecture tools into one common design environment. It also integrates SOC simulation and analysis tools. Xtensa Xplorer is a visual environment with a host of automation tools that makes creating Xtensa processor-based SOC hardware and software much easier.

Xtensa Xplorer supports editing C/C++ and assembly as well as TIE, allowing users to create their TIE applications with syntax-aware editing and automatic support for building, estimating, and running applications that use TIE.

Xtensa Xplorer can automatically generate and manage makefiles. It provides a graphical multiprocessor debugger and extensive performance visualization tools (see Section 5.2.2 on page 24 for more information about the Xtensa Xplorer Debugger).

Xtensa Xplorer supports the creation, programming and debugging of shared memory multiprocessor subsystems.

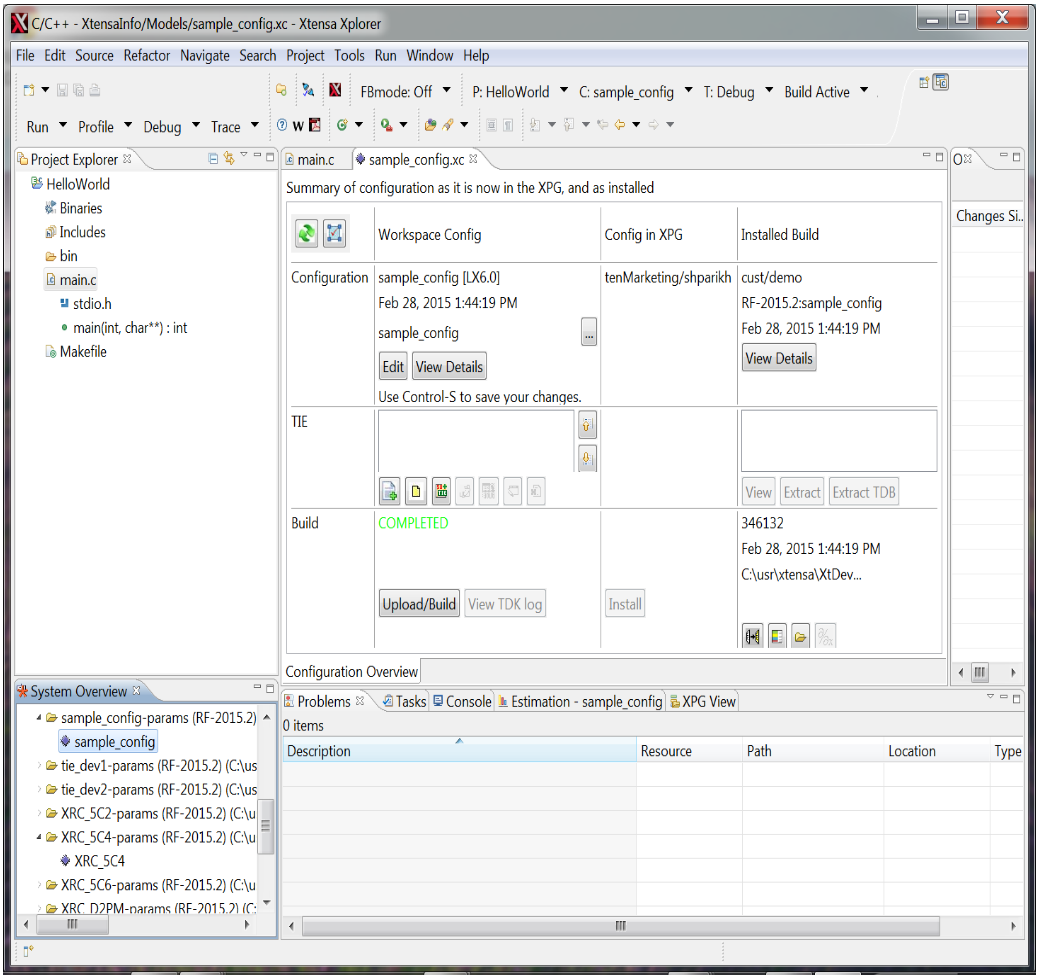
2.1 Downloading Xtensa Tools in Xtensa Xplorer

Xtensa Xplorer downloads and unpacks or unzips the file containing the version of Xtensa Tools you select. Because the downloaded archive file remains in the `download` directory, you can transfer it to other systems for installation without having to download it again. You can easily share the same Xtensa Tools for development with both Xtensa Xplorer and a command-line environment.

Linux users who download configurations for command-line installation can install those same configurations into Xtensa Xplorer by directing the New Configuration wizard to the same download file (for example, `s1_tar.Z`). For detailed instructions to download Xtensa Tools using Xtensa Xplorer, refer to the *Xtensa Development Tools Installation Guide*.

2.2 Editing and Installing Configuration Builds in Xtensa Xplorer

To install a configuration build, you first configure and build a processor with options of your choice, and then upload and install that configuration. Figure 2 illustrates the Xtensa Xplorer's System Overview window, where you select an existing configuration or create a new configuration.



Summary of configuration as it is now in the XPG, and as installed

	Workspace Config	Config in XPG	Installed Build
Configuration	sample_config [LX6.0] Feb 28, 2015 1:44:19 PM sample_config Edit View Details Use Control-S to save your changes.	tenMarketing/shparikh	cust/demo RF-2015.2:sample_config Feb 28, 2015 1:44:19 PM View Details
TIE			View Extract Extract TDB
Build	COMPLETED Upload/Build View TDK log	Install	346132 Feb 28, 2015 1:44:19 PM C:\usr\xtensa\XtDev...

System Overview:

- sample_config-params (RF-2015.2)
  - sample_config
- tie_dev1-params (RF-2015.2) (C:\us
- tie_dev2-params (RF-2015.2) (C:\us
- XRC_5C2-params (RF-2015.2) (C:\u
- XRC_5C4-params (RF-2015.2) (C:\u
  - XRC_5C4
- XRC_5C6-params (RF-2015.2) (C:\u
- XRC_D2PM-params (RF-2015.2) (C:

0 items

Description	Resource	Path	Location	Type

Figure 2. Xtensa Xplorer Window

After you install a configuration, you can edit, build and re-install it for performance tuning. When you edit a configuration, the configuration information is saved in the workspace directory.

Figure 2 also shows the eight configuration tabs at the bottom of the Summary window. Select a tab to open the corresponding configuration options, and select the options for your configuration.

Refer to the *Xtensa Development Tools Installation Guide* for detailed procedures to install and edit configuration builds in Xtensa Xplorer. For more detailed instructions on how to use the Xtensa Tools in the Xtensa Xplorer environment, refer to the Xtensa Xplorer documentation.

3. Command-Line Environment Setup

Once you have followed the installation procedure and installed a configuration (either using Xtensa Xplorer or the standalone command-line mode) the Xtensa processor is ready for your design development (refer to the *Xtensa Development Tools Installation Guide* for installation directions). This chapter applies to the command-line mode usage and its environment setup.

3.1 Xtensa Tools Environment Setup

For Linux hosts, add the bin directory of the Xtensa Tools to your path. In the following commands, `<xtensa_tools_root>` is the directory where the Xtensa Tools have been installed.

- For csh and derivatives, set your PATH environment variable to:

```
set path = (<xtensa_tools_root>/bin $path)
```

- For sh and derivatives, set your PATH environment variable to:

```
export PATH; PATH=<xtensa_tools_root>/bin:$PATH
```

For Windows, select **Xtensa Configuration/CMD Shell** from the Windows **Start/Programs** menu to automatically set the path. The paths are only set correctly if you run the tools from the selected command shell.

3.2 Selecting a Processor Configuration

When you run the Xtensa Tools, you must specify which Xtensa configuration you want to use. For Windows, selecting the appropriate **Xtensa Configuration/CMD Shell** from the Windows **Start/Programs** will automatically select the appropriate configuration.

To select a configuration on Linux, use either the `––xtensa-core` command-line option or the `XTENSA_CORE` environment variable. For example, using a C shell, the following commands are equivalent:

- `xt-xcc -c --xtensa-core=vectra_le myprog.c`
- `setenv XTENSA_CORE vectra_le; xt-xcc -c myprog.c`

Note: If both variables are specified, the command-line option takes precedence over the environment variable.

Xtensa customers who use only one Xtensa configuration can avoid having to specify which configuration to use. When you install a configuration, the installation program gives you the option of making it the default configuration. If you select this configuration as the default, the installation program adds an entry to the Xtensa core registry with the name `default`. If you do not specify a configuration name when you run the tools, the tools will look in the registry for a `default` configuration.

The install program for an Xtensa configuration adds the configuration to an Xtensa core registry. The default registry is the `<`*`xtensa_tools_root`*`>/config` directory. This one default registry is usually sufficient, but Xtensa Tools allow you to set up as many Xtensa core registries as you need. Multiple registries may be helpful, for example, when different groups of users are sharing the same installation of Xtensa Tools, but each group has its own set of Xtensa configurations. When working on multiple processor systems, you may also want to have a separate registry for each set of Xtensa configurations that are used together in a system. “Managing Xtensa Core Registries” on page 12 describes how to set up additional registries. To specify an alternate Xtensa core registry when running Xtensa Tools, you can use either the `--xtensa-system` command-line option, or the *`XTENSA_SYSTEM`* environment variable. For example:

- `xt-run --xtensa-system=~/xtregistry myprog`
- `setenv XTENSA_SYSTEM ~/xtregistry; xt-run myprog`

Both of these commands cause the `xt-run` simulator to use the `~/xtregistry` directory as the Xtensa core registry. The command-line option takes precedence over the environment variable.

Instead of specifying a single registry, you can also specify a semicolon-separated list of registries to search. The first registry in this list with a matching core name will then determine the Xtensa configuration to be used. For example, if you want to search two registries, you might use something similar to one of the following commands:

- `xt-run --xtensa-system="~/xtregistry;/xtensa_tools/config" myprog`
- `setenv XTENSA_SYSTEM "~/xtregistry;/xtensa_tools/config"; xt-run myprog`

Although using a list of registries may be convenient at times, it can also be confusing as it is harder to keep track of which configuration is being used. An alternative is to set up a new registry and copy into it the entries for the Xtensa configurations that you want to use.

3.3 Specifying a TIE Development Kit

The Instruction Extension (TIE) compiler allows you to dynamically extend a configured Xtensa processor without having to return to the Xtensa Processor Generator. When you run the TIE compiler, it creates a TIE Development Kit (TDK) directory that the Xtensa Tools can use to update an Xtensa configuration.

After you create a TDK directory, you need to direct the Xtensa Tools to use it. Use either the `––xtensa-params` command-line option or the *`XTENSA_PARAMS`* environment variable to direct this. For example, the following two commands both run the assembler with the TDK directory set to `~/mytdk`:

- `xt-as --xtensa-params=~/mytdk asmfile.s`
- `setenv XTENSA_PARAMS ~/mytdk; xt-as asmfile.s`

3.4 Querying Configuration Settings

Xtensa Tools provide an option to query the Xtensa core registry. This may be helpful if you are unsure which Xtensa configuration is being used. It also provides a way to programmatically locate various Xtensa files from scripts and makefiles. All of the Xtensa Tools that read the Xtensa core registry accept a `––show–config=<`*`info`*`>` option to display various kinds of configuration information. Table 3 shows the information that can be retrieved with different values of `<`*`info`*`>`.

Table 3. Values for the --show–config=<*info*> Command-Line Option

<*info*> Value	**Description**
`cores`	List of available Xtensa configurations
`cores-raw`	Same as `cores`, except it does not print the registry directory information
`core`	The selected entry from an Xtensa core registry
`tdks`	TIE development kit (TDK) directory
`tools`	The Tools sub-directory under the <*`xtensa_tools_root`*> directory
`tctools`	TIE compiler directory
`xttools`	The <*`xtensa_tools_root`*> directory
`config`	The directory where the configuration-specific core files have been installed
`all`	All of the information above
`help`	This list of values for the `––show–config` option

Xtensa Tools also provide a way to query an Xtensa object or binary file for properties about the configuration that was used to build the object or binary file. This is helpful in preventing the user from using the wrong configuration.

- `xt-config file.o`

3.5 Managing Xtensa Core Registries

As an Xtensa core registry is nothing more than a directory containing certain files, it is easy to create and modify a core registry. Each registry directory contains a parameter file for each configured Xtensa processor. The parameter file for a configuration named `<config>` has a file name of `<config>–params`. You can copy a parameter file from one registry to another. You can also get a copy of the parameter file for an Xtensa configuration from its `<xtensa_root>/config` directory, with the file name of `default-params`. Here `<xtensa_root>` is the directory where the configuration-specific core files have been installed.

To create a new Xtensa core registry, create a new directory and copy a set of Xtensa parameter files into it. Then set `XTENSA_SYSTEM` or specify the `––xtensa-system` option to make the Xtensa Tools use the new registry. You can add, remove and rename entries in an Xtensa core registry by adding, removing and renaming the corresponding parameter files.

To set up a default configuration, add a `default-params` file to the core registry. On Linux systems, as a convention, this file should be a symbolic link to another parameter file. You can determine the current default configuration by examining the `default-params` link.

Refer to the *Xtensa Development Tools Installation Guide* for more information about Xtensa core registries.

3.6 Software Development Flow

The rest of this guide presents highlights of the Xtensa software development tools and how they are used in a typical software development flow. Once a program is written, it is compiled and run on the processor or simulator. The compiler, assembler and linker generate the binary that is loaded and run. The debugger debugs the program and the profiler and simulator tune the performance of the program.

The following example program illustrates the software development process:

```
/* An example program */

#include <stdio.h>

#define NUM_ELEMENTS 100
```

```
#define NUM_ITERS    1000

/* Simple array computation */
void calc_array(int *x)
{
  int i, iters;

  for (i = 0; i < NUM_ELEMENTS; i++)
    x[i] = 1;

  for (iters = 0; iters < NUM_ITERS; iters++)
    for (i = 1; i < NUM_ELEMENTS; i++)
      x[i] += x[i-1];
}

main()
{
  int a[NUM_ELEMENTS];
  calc_array(a);
  printf("Done computing the array\n");
}
```

This example program is stored as `examples/ex1.c` in your configuration-specific installation directory.

4. Command-Line Driven Compiling, Assembling and Linking

As shown in Figure 3, the first development step is to write or modify an application, which includes compiling, assembling and linking the program. This chapter discusses each of these tasks.

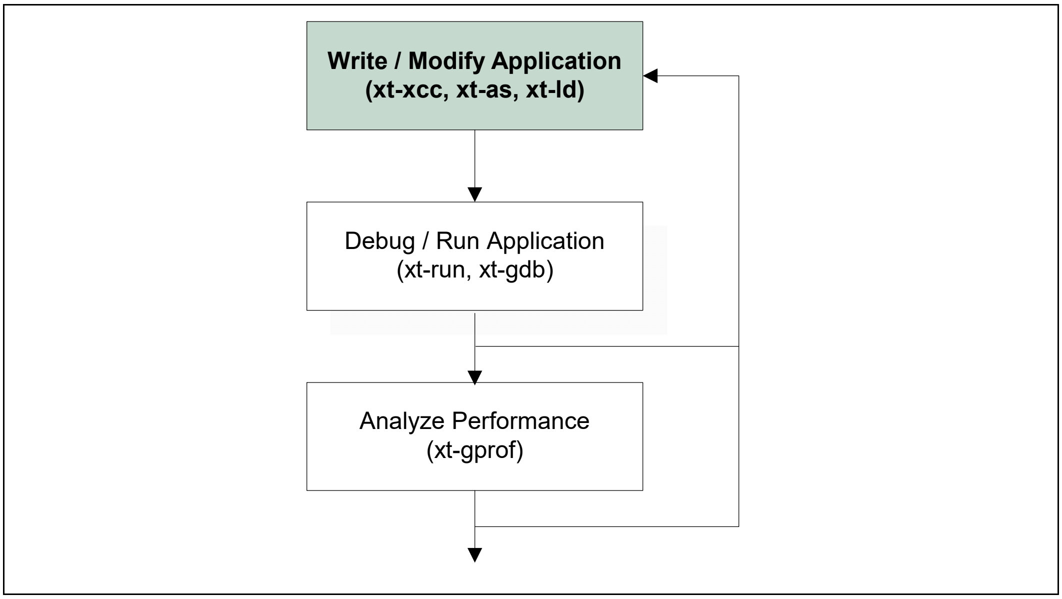


Figure 3. Creating the Application

4.1 Compiling

The Xtensa C and C++ Compiler (XCC) is an optimized compiler for the Xtensa processor. Operation of XCC is similar to operation of standard GNU C and C++ compilers. XCC provides superior code execution performance and reduced code size through improved optimization and code generation technology. Refer to the *Xtensa C and C++ Compiler User's Guide* for details about this compiler.

To generate an executable from the `ex1.c` example program (see Section 3.6 “Software Development Flow” on page 12), enter the following command:

```
xt-xcc -O2 ex1.c -o ex1
```

The above command contains the `-o` option, which specifies the name of the executable output file. It also uses the `-O2` option, which compiles with additional optimizations.

Invoke the compiler using `xt-xcc --help` to get a brief description of all the options supported by the compiler.

Starting with RI-2018.0, the SDK also includes the new Xtensa XT-CLANG compiler (`xt-clang` for C and `xt-clang++` for C++), which is an implementation of the LLVM Clang C/C++ compiler for Xtensa processors. Refer to the *Xtensa XT-CLANG Compiler User's Guide* for details.

4.2 Assembling

The GNU assembler translates Xtensa assembly source files into machine code using a standard ELF object file format. You can invoke the Xtensa assembler `xt-as` directly or indirectly, through the compiler driver (for example, `xt-xcc`).

The compiler driver recognizes source files with names ending in `.s` or `.S` as assembly files and invokes the assembler to translate them to object files. The advantage of using the compiler driver with assembly files is that it is easy to run the files through the C preprocessor before assembling them. If the assembly source file ends with an uppercase S (`.S`), the compiler driver automatically invokes the C preprocessor. This allows you to use C-style comments, macro expansion and header file inclusion in your assembly files. We provide a number of header files to define macros that are useful in assembly code.

The assembler's instruction scheduler utilizes its knowledge of the pipeline and data dependences to reorder or substitute instructions to avoid pipeline stalls. For Xtensa LX processors, it also automatically bundles operations into Flexible Length Instruction eXtension (FLIX) instructions. Use the `--schedule` flag to enable this feature. Scheduling is useful for hand-written assembly code. For details about the assembler's scheduler, see the *GNU Assembler User's Guide*.

4.3 Linking

After compiling and assembling your source files, you must link the resulting object files together to create the application program. As with the assembler, it is generally a good idea to invoke the `xt-ld` linker via the compiler driver. The compiler driver automatically specifies to the linker various system object files, default libraries, and linker options. It is much simpler to let the compiler do this than to invoke the `xt-ld` linker directly with all the right arguments. This is especially true for C++ programs. If the application includes any C++ code, be sure to link using the C++ compiler driver (`xt-xc++`).

The linking process depends on the system memory layout (for example, where the program should reside in memory) and the system software or operating system (such as what start-up files to include in the application and what libraries to include).

We have introduced linker support packages (LSPs) to specify these linker dependencies and to make it easy to switch between different environments. For example, you can use different LSPs for developing your application running on the Xtensa ISS, running hardware diagnostics, debugging on an emulation kit board, and creating the final software for the product. Cadence provides some default LSPs with every Xtensa processor configuration and you can also create your own custom LSP. To specify which LSP you wish to use, link with the `-mlsp` compiler option. The default is the `sim` LSP, which supports semi-hosting on the ISS.

Semi-hosting allows Xtensa executables to perform some common system calls in the simulator's host environment; for example, the Xtensa program can read and write files on the simulator's host machine.

For software development using a development board connected to an Xtensa debugger via the OCD Daemon, you can get similar functionality using the `gdbio` LSP by linking with `-mlsp=gdbio`. The target program will use the host file system for standard I/O functionality. Data will automatically be transferred between the host and target using the debugger.

For details about LSPs, see the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

5. Simulation, Debugging, and Profiling

This chapter describes how to run the simulator, debugger and profiler.

After creating the application program, the next step is to run the program on a simulator and use a debugger to help locate any problems. This step is shown in Figure 4.

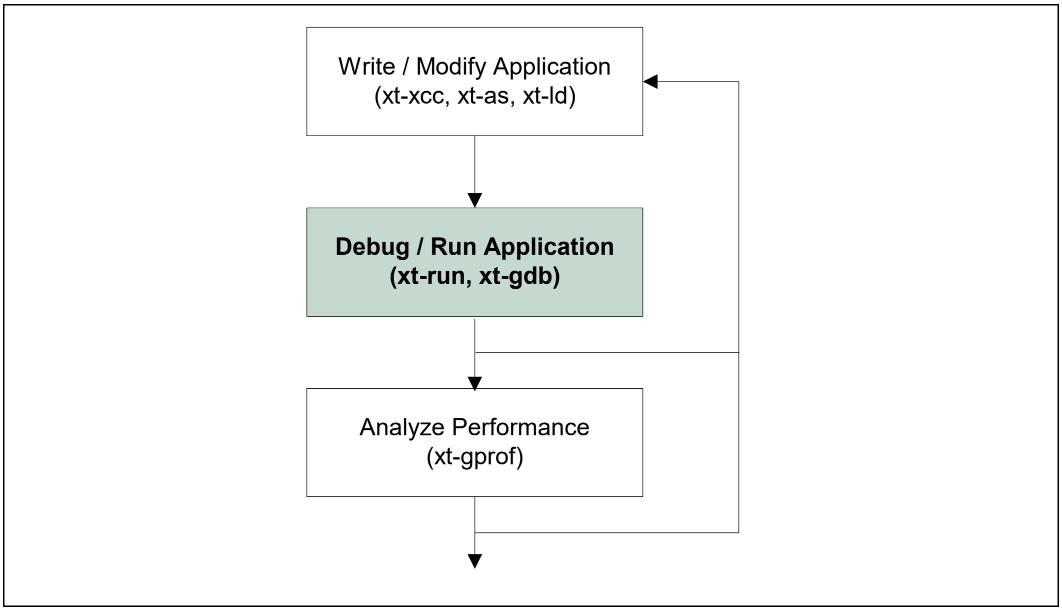


Figure 4. Debugging the Application

5.1 Running the Simulator

Once the program has been compiled, assembled and linked successfully, it is ready to be executed. The program can be executed on the Xtensa Instruction Set Simulator (ISS), which is invoked as `xt-run`. The syntax for invoking the ISS is:

```
xt-run [simulator-options] target-program [target-arguments]
```

The command for simulating our example program is:

```
xt-run ex1
```

For the example program, the simulator responds with:

```
Done computing the array
```

The simulator has a number of command-line options. Brief descriptions of these options can be found by invoking the simulator with:

```
    xt-run --help
```

Table 4 lists the ISS command-line options that are of particular interest.

Table 4. Simulator Command-Line Options

Command	Description
`--summary`	Displays a brief performance summary of the simulator run.
`--mem_model`	Enables memory subsystem modeling (see Section 5.1.1 for more information).
`--turbo`	Enables fast functional simulation (see Section 5.1.2 for more information).
`--profile=`*name*	Generates profiling data to the named file (see section 5.3 for more information).

For details about the simulator, see the *Xtensa Instruction Set Simulator (ISS) User's Guide.*

5.1.1 Memory Modeling

The `--mem_model` option listed in Table 4 enables the simulation of internal memories (instruction cache, data cache, local RAMs/ROMs) and a parameterized simple delay model of system memories (attached to the PIF). The following example shows the simulator output using the `--mem_model` option:

```
$ xt-run --mem_model ex1
Done computing the array

Xtensa Core: "s1_config"  ISS Version: Xtensa 10.0.0

Time for Simulation = 0.99 seconds (user = 0.99 system = 0.00)

current pc = 0x40007907

Cache Configuration:
  I cache: 8192 bytes (8KB), direct-mapped, 16-byte line
  D cache: 8192 bytes (8KB), direct-mapped, 16-byte line

Events                           Number  Number
                                         per 100
                                         instrs

Committed instructions          337241 ( 100.00 )
Instruction fetches             239865 (  71.13 )
   Uncached                       1700 (   0.50 )
   ICache fetches               238165 (  70.62 )
      ICache misses                245 (   0.07 )  0.10% of ICache fetches
Taken branches                    1500 (   0.44 )
Exceptions                           4 (   0.00 )
   WindowOverflow                    2 (   0.00 )
   WindowUnderflow                   2 (   0.00 )
Loads                           114337 (  33.90 )
   Uncached                          6 (   0.00 )
   DCache loads                 114331 (  33.90 )
      DCache load misses           104 (   0.03 )  0.09% of DCache loads
Stores                           99264 (  29.43 )
   Cached                        99264 (  29.43 )
      DCache write-thru misses     213 (   0.06 )  0.21% of cached stores
PIF Transfers (4 bytes each)    102361 (  30.35 )
   IFetch reads                   2675 (   0.79 )
   Data reads                      422 (   0.13 )
   Data writes                   99264 (  29.43 )

Cycles: total = 357477
                                                 Summed |          Summed
                                         CPI      CPI   |% Cycle  % Cycle
Committed instructions        337241 ( 1.0000   1.0000 |  94.34    94.34 )
Taken branches                  3022 ( 0.0090   1.0090 |   0.85    95.18 )
Pipeline interlocks              135 ( 0.0004   1.0094 |   0.04    95.22 )
ICache misses                   2227 ( 0.0066   1.0160 |   0.62    95.85 )
DCache misses                   1012 ( 0.0030   1.0190 |   0.28    96.13 )
Exceptions                        20 ( 0.0001   1.0190 |   0.01    96.13 )
Uncached ifetches               8864 ( 0.0263   1.0453 |   2.48    98.61 )
Uncached loads                    37 ( 0.0001   1.0454 |   0.01    98.62 )
Sync replays                    2630 ( 0.0078   1.0532 |   0.74    99.36 )
Special instructions             280 ( 0.0008   1.0540 |   0.08    99.44 )
Loop overhead                   2004 ( 0.0059   1.0600 |   0.56   100.00 )
Reset                              5 ( 0.0000   1.0600 |   0.00   100.00 )

Cycles: total = 357396
```

Table 5 describes the sections in the ISS summary output.

Table 5. Simulator Summary Output Sections

Section/Column	Description
`Time for Simulation`	Shows the amount of host CPU time it took to simulate the program.
`Cache Configuration`	Provides details about the instruction and data cache for the particular Xtensa processor configuration being simulated.
`Events`	Provides statistics about different events that occurred in the target program.
`Number`	Gives the absolute count for each event type.
`Number per 100 instrs`	Divides each event count by the total number of successfully executed instructions and displays the amount as a percentage.
`Cycles`	Shows the total cycle count for the target program and contributions of various events. There are five columns for each event category: ▪ The first column is the absolute cycle count. ▪ The second column is the event's contribution to the CPI (Cycles Per Instruction). ▪ The third column is the running sum of the second column. ▪ The fourth column is the percentage of total cycles attributable to the event. ▪ The last column is the running sum of the fourth column.

5.1.2 Fast Functional Simulation (TurboXim)

The default simulation mode in the Xtensa ISS is the pipeline-accurate simulation. In this mode, the simulator models the Xtensa processor micro-architecture with a very high degree of cycle accuracy. By using the `--turbo` option, you can change the simulation mode to the fast functional simulation or TurboXim. In TurboXim mode, the ISS provides architecturally correct simulation of the Xtensa instruction set, while achieving a simulation speed that is 40-80 times higher than in the pipeline-accurate mode.

For differences between the pipeline-accurate and the fast functional simulation modes, see the *Xtensa Instruction Set Simulator (ISS) User's Guide.*

5.1.3 Ferret Memory Checker

The ISS ferret client checks for some common memory problems that programming errors can cause. Ferret can help you debug memory errors, such as freeing a heap-allocated block of memory more than once, or the collision of heap and stack. To enable this feature from the command line, link your target program to the ferret library (`-lferret`), which supplies special versions of *`malloc`*, *`free`*, and *`sbrk`* that support this program checking. Then invoke the simulator with the `--client_cmds=ferret` option or when using *xt-gdb*, enter the command `target sim --client_cmds=ferret`. Xplorer users can select ferret from the *Memory* tab of the *Project/Xtensa Project Build Properties* menu and then enable ferret from the *Advanced* tab of the run or debug launch.

Ferret runs on the simulator and generates warnings for each memory error detected. Warning messages include the address of the instruction that caused the fault, the memory address associated with the fault, and the type of fault. When run from within a debugger, the debugger will break on every ferret warning. Refer to the *Xtensa Instruction Set Simulator (ISS) User's Guide* for more information about using the ferret client.

5.1.4 System-Level Modeling (XTMP and XTSC)

For modeling system-on-chip (SOC) designs, which often contain multiple processors and various other devices, we provide two application programming interfaces to the ISS: Xtensa Modeling Protocol (XTMP) and Xtensa SystemC package (XTSC). You can use XTMP or XTSC to write your own customized, multi-threaded simulators to model complex SOCs. This allows you to develop, debug, profile, and verify combined hardware and software systems early in the design process. In addition, pre-built binaries, *`xtmp-run`* and *`xtsc-run`*, are provided to allow you to simulate subsystem made up of pre-defined components without the need to write your own simulation models.

XTMP is Cadence's original C-level API for system-level modeling.
Note: The XTMP system-modeling API is not supported for configurations based on the Xtensa NX architecture.

XTSC provides a SystemC model of the Xtensa processor and a well-defined set of TLM (transaction-level modeling) interfaces for communication with SystemC models of other SOC components. For details about XTSC, see the *Xtensa SystemC User's Guide*.

5.2 Running the Debugger

If there are runtime errors, you can execute the program in the debugger. Xtensa software tools include two debuggers:

- The GNU command line debugger (`xt-gdb`)
- The debugger in the Xtensa Xplorer environment

These two debuggers can debug a program running on an Instruction Set Simulator (ISS) or on a hardware platform with an On-chip debugging (OCD) interface.

5.2.1 `xt-gdb` Debugger

The `xt-gdb` debugger uses the Instruction Set Simulator (ISS) to run your program, unless you specify another target. For example, to launch the debugger and run the example program on the simulator, the session looks like this:

```
xt-gdb ex1
```

The debugger responds, displaying the version and copyright notice, followed by this `xt- gdb` command-loop prompt:

```
GNU gdb <version>
... copyright and configuration notice ...
(xt-gdb)
```

To run the program on the simulator, enter the debugger's `run` command, as follows:

```
(xt-gdb) run
Starting program: ex1
Remote debugging using localhost:58733
Done computing the array
(xt-gdb)
```

The debugger automatically launches the simulator when you enter the `run` command at the debugger's command-loop prompt. You can specify simulator options using the debugger's `target iss` command before entering the debugger's `run` command. The debugger runs the simulator by default, unless you specify another target.

For example, a debugger session that specifies simulator options may look like this:

```
$ xt-gdb ex1
GNU gdb <version>
... copyright and configuration notice ...
(xt-gdb)
(xt-gdb) target iss --mem_model --pchistory=10
(xt-gdb) run
... debug summary ...
(xt-gdb)
```

For details on the debugger, see the *GNU Debugger User's Guide*. The simulator options are described in the *Xtensa Instruction Set Simulator (ISS) User's Guide.*

5.2.2 Xtensa Xplorer Debugger

The Xtensa Xplorer debugger provides an easy to use, graphical environment for debugging your software. The Xplorer debugger provides visual inspection of variables, registers, and expressions and easy control of breakpoints, watchpoints and other debugger features. Using *TRAX* hardware or a simulator target, when the target is suspended, Xtensa Xplorer can display the last *N* executed instruction, giving the user a window back in time. Xtensa Xplorer has a host of additional support for debugging multiple processor applications either through XTSC/XTMP or with an XOCD connection to real hardware. The Xtensa Xplorer Debugger can debug software developed both inside and outside the Xplorer IDE.

In all modes, Xplorer accepts command-line options and environment variables, as specified in Section 3.2, Section 3.3, and Section 3.4, to specify and display the Xtensa core registry, configuration and TIE development kit.

Using Xtensa Xplorer Debugger as Part of the Xplorer IDE

To use Xplorer to debug software developed using Xplorer as a complete IDE, refer to the documentation in the Xtensa Xplorer download for information about the features and detailed instructions.

Using Xtensa Xplorer Debugger to Debug Software Outside of the IDE

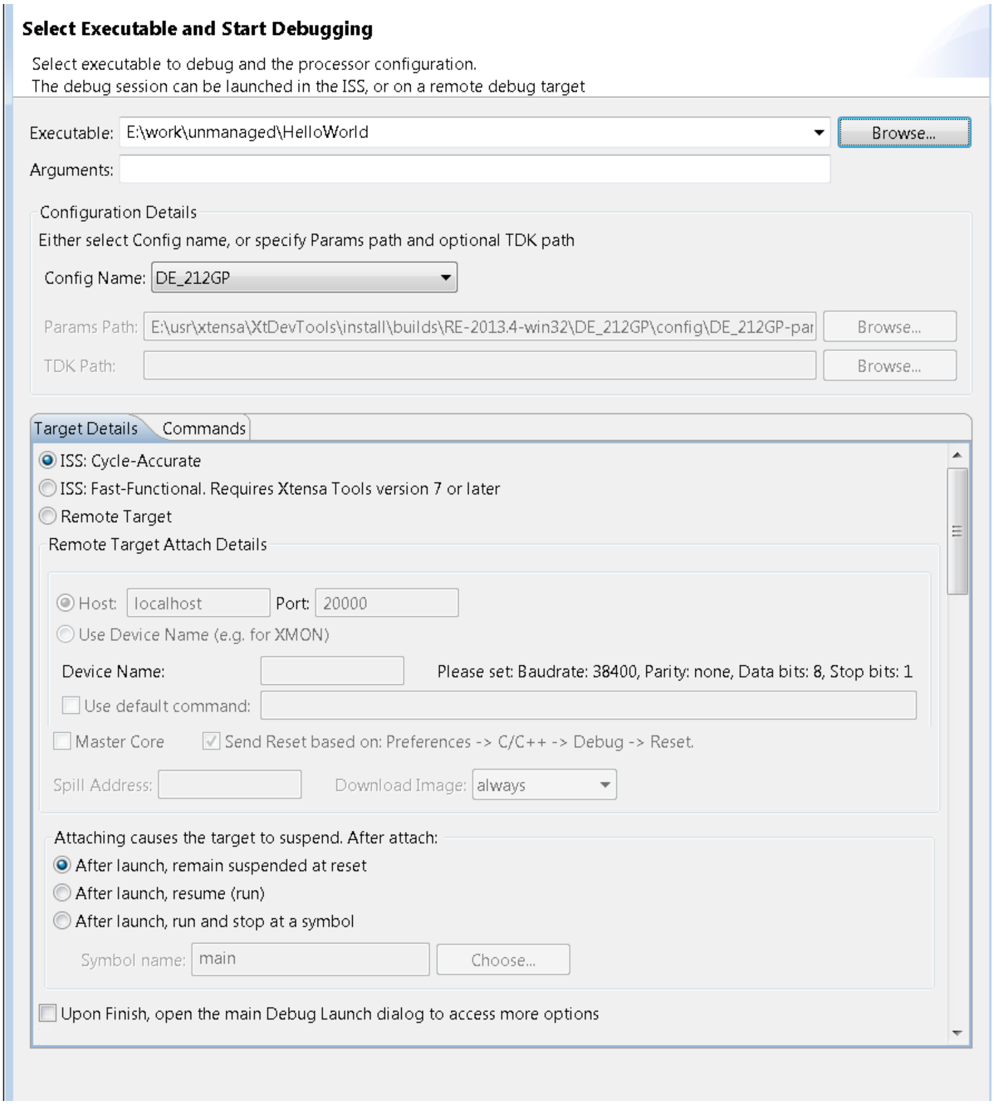
Users who prefer to use the Xtensa command-line tools to develop their software can still use Xplorer's very convenient debugging environment. To use Xplorer to debug an application compiled and linked outside of the IDE, use the following command, which launches Xplorer in standalone debugging mode:

```
xplorer [xplorer_args] --debug program [program_args]
```

`program` can be any Xtensa program compiled with the command-line tools. *`xplorer-args`* can be one of *`--xtensa-system`*, *`--xtensa-core`*, *`--xtensa-params`*, *`--xtensa-show-config`* or *`--srcpath`*. The option `--srcpath` *`dirlist`* is used to specify the directories containing source files. The directories in *`dirlist`* should be separated by colons (on Linux) or semicolons (on Windows).

The program need not have been compiled in the graphical environment, but should have had `-g` specified at compile time if symbolic debugging is desired. If no workspace is specified (using `-data` *`workspace`* as the first argument), `xplorer` creates a temporary directory for the workspace. If a workspace is specified, it must not be a sibling of or subordinate to any of the source directories.

When Xtensa Xplorer starts up in standalone debugging mode, the Select Executable and Start Debugging dialog box opens as shown in Figure 5, allowing you to specify the debugging session's settings. Click Finish to debug the executable.



Select Executable and Start Debugging

Select executable to debug and the processor configuration.
The debug session can be launched in the ISS, or on a remote debug target

Executable: E:\work\unmanaged\HelloWorld

Arguments:

Configuration Details

Either select Config name, or specify Params path and optional TDK path

Config Name: DE_212GP

Params Path: E:\usr\xtensa\XtDevTools\install\builds\RE-2013.4-win32\DE_212GP\config\DE_212GP-par

TDK Path:

Target Details | Commands

[✓ ISS: Cycle-Accurate]
ISS: Fast-Functional. Requires Xtensa Tools version 7 or later
Remote Target

Remote Target Attach Details

[✓ Host:] localhost Port: 20000
Use Device Name (e.g. for XMON)

Device Name: Please set: Baudrate: 38400, Parity: none, Data bits: 8, Stop bits: 1

Use default command:

Master Core [✓ Send Reset based on: Preferences -> C/C++ -> Debug -> Reset.]

Spill Address: Download Image: always

Attaching causes the target to suspend. After attach:

[✓ After launch, remain suspended at reset]
After launch, resume (run)
After launch, run and stop at a symbol

Symbol name: main

Upon Finish, open the main Debug Launch dialog to access more options

Figure 5. Xplorer Standalone Debugging Mode Dialog Box

Many of the options in this dialog box are common to both the ISS and the remote target mode. The options are

- **Executable**: To specify your executable. Cadence recommends using the command line to set the executable.
- **Arguments**: To specify arguments to the executable. Cadence recommends using the command line to set the arguments.

- **Config Name**: The default value, **None**, is typical for the standalone debugging process. However, if you specify an executable that is compiled in Xtensa Xplorer, it will display a configuration name.
- **Params Path**: Set from the environment, or from the `--xtensa-system`, `--xtensa-core` command-line arguments, or derived from the pathnames embedded in the executable. You should not need to modify these entries, but check them for correctness before launching the program.
- **Target Details: Select ISS or Remote Target**: If you select **ISS**, you must choose between Cycle-Accurate or Fast-Functional mode. If you select **Remote Target**, you also need to specify the following information related to the remote target:
  - **Host:Port**: The hostname (or IP address) and the port number of the machine running XOCD, XTSC or XTMP; default: `localhost:20000`.
  - **Spill Address**: `xt-gdb` requires some scratch memory to retrieve the contents of TIE registers. The contents of this area are saved and restored, so it can safely overlap either program text or data. The only constraints are that it must be writable and properly aligned (16-byte alignment is sufficient). In most circumstances, the spill address argument is not required; `xt-gdb` computes a safe default.
  - **Clear processor state when downloading image**: Downloading an image to an emulator requires clearing a few processor states, so you need to select this option when downloading an image to a processor emulator. The PS (program status) register is set to 1 and the INTENABLE (interrupt enable) state is set to 0 when you select this option.
  - **Download Image**: You can choose to have Xtensa Xplorer always download the image, never download the image, or ask you before downloading the image.
- **Attaching causes the target to suspend. After attach**: After the Xtensa Xplorer makes a connection to the target, the target is suspended. Use this option to specify what should happen at that point. The target may remain in a suspended state, resume and stop in `main`, or simply resume.

Selecting the Commands tab presents a multi-line field where you can type in additional commands to issue to xt-gdb at the launch. This is especially useful for a remote launch (*e.g.*, hardware target) where either some additional setup is required before the application can be downloaded, and / or some special settings need to be made before it should be run. If in the launch you let Xplorer download the binary, then by default the commands are run before the launch. If you change the *download image* setting to prevent Xplorer downloading, then you can specify a complete sequence of commands that includes the launch under your control.

Appropriate xt-gdb commands to issue are those which affect the target (*e.g.,* writing to memory). You should not issue commands which might conflict with UI behavior that is controlled through the IDE.

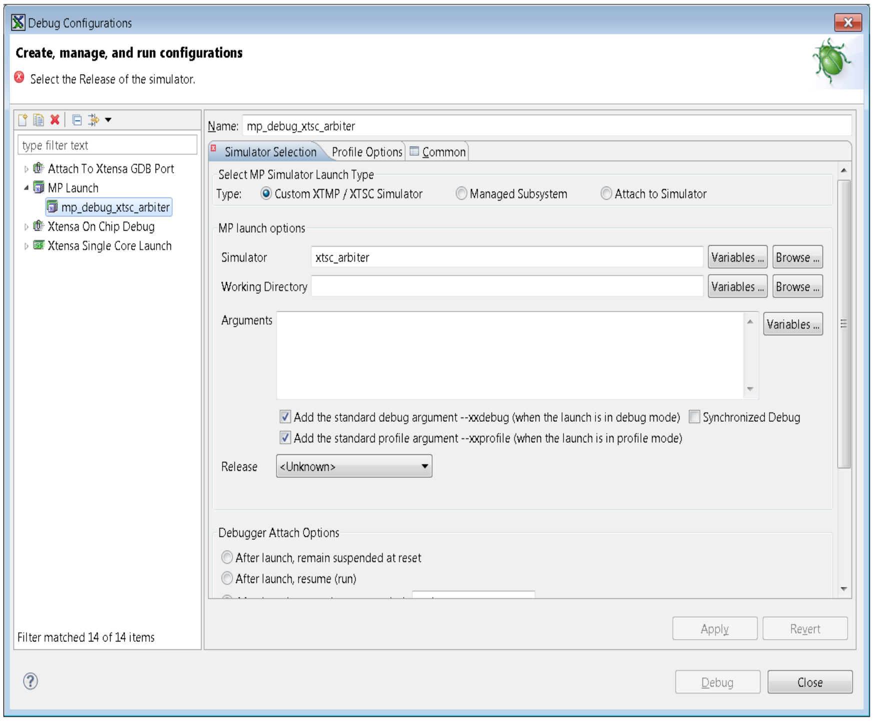
Using Xtensa Xplorer to Debug Multi-processor Simulations Outside of the IDE

To use Xtensa Xplorer to debug multi-processor simulations, use the following command, which launches Xplorer in multi-process debugging mode:

```
xplorer [xplorer_args] --mp-debug simulator [simulator_args]
```

In this mode, `xplorer` also accepts `--srcpath` *`dirlist`* to specify the directories containing target source files. The directories in *`dirlist`* should be separated by colons (on Linux) or semicolons (on Windows). The *`simulator`* argument is the pathname of an XTMP or XTSC simulator program; see the *Xtensa® Modeling Protocol (XTMP) User's Guide* or the *Xtensa SystemC User's Guide* for details. Xtensa Xplorer will start the simulator and attach to each separate target program that was enabled for debugging. You can then control the target programs via the graphical debugger.

When Xtensa Xplorer starts up in multi-process debugging mode as shown in Figure 6, the Select Simulator and Launch dialog box opens, allowing you to specify the pathname to the simulator executable, the working directory, and the simulator arguments. You can also adjust the timeout for debugger connections and the session start behavior.


Debug Configurations

Create, manage, and run configurations

Select the Release of the simulator.

type filter text
- Attach To Xtensa GDB Port
- MP Launch
  - mp_debug_xtsc_arbiter
- Xtensa On Chip Debug
- Xtensa Single Core Launch

Filter matched 14 of 14 items

Name: mp_debug_xtsc_arbiter

Simulator Selection | Profile Options | Common

Select MP Simulator Launch Type

Type: [● Custom XTMP / XTSC Simulator] ○ Managed Subsystem ○ Attach to Simulator

MP launch options

Simulator: xtsc_arbiter

Working Directory:

Arguments:

[✓] Add the standard debug argument --xxdebug (when the launch is in debug mode) [ ] Synchronized Debug

[✓] Add the standard profile argument --xxprofile (when the launch is in profile mode)

Release: <Unknown>

Debugger Attach Options

○ After launch, remain suspended at reset

○ After launch, resume (run)

Figure 6. Xplorer Multi-Process Debugging Mode Dialog Box

Using Xtensa Xplorer to Debug Multiple Processors via OCD

Inside the Xplorer IDE, select an MP launch to simultaneously debug multiple processors. The processors can run in synchronized mode where all processors stop, step and continue within a few cycles of each other or the processors can run unsynchronized, allowing you to debug one processor while others continue running freely.

Using the standalone tools, invoke

```
xplorer [xplorer_args] --debug program [program_args]
```

individually on each processor that you wish to debug. Make sure to use a separate workspace for each concurrently running copy of Xtensa Xplorer.

5.3 Extracting Information From Trace Signals

Many bugs on hardware targets are difficult to reproduce in a debugger, either because they require real-time system interactions or because they involve race conditions that are not easily reproducible. For such problems, you can analyze program trace information to debug problems.

Xtensa processors can be configured with a trace port. The direct output of the trace port can be fed to the `xt-trace` software tool that converts raw trace signal information into a stream of assembly instructions. It also offers varying levels of processor-state information. Details about the `xt-trace` tool can be found in the *Xtensa Debug Guide.*

We also offer a TRAX-PC compressor module as an optional extra. TRAX-PC compresses raw program counter information and stores it into a local, on-chip, buffer. For command line users, the `xt-traxcmd` tool controls the trace module and allows the setting of triggers. The `xt-traxview` tool converts the compressed trace files into a series of annotated assembly instructions. Details can be found in the *Xtensa Debug Guide*.

Xplorer brings together these two tools and the debugger in a unified framework as shown Figure 7.

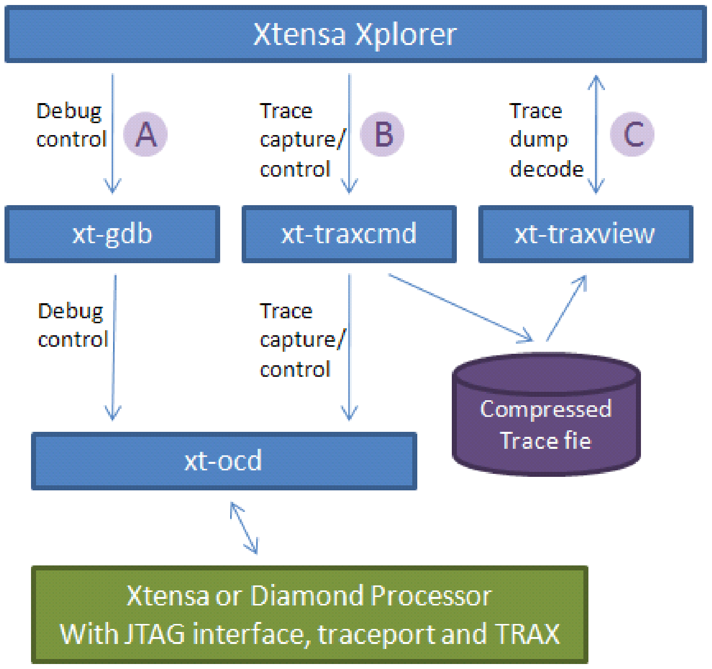


Figure 7. Xtensa Xplorer Unified Framework

If you are driving and gathering trace data from a hardware target with Xplorer but are not debugging (that is, a *trace* launch), then Xplorer is driving `xt-traxcmd` which, in turn, drives `xt-ocd` but in a non-intrusive way. If you are using Xplorer as a hardware target debugger and are also collecting trace data, then Xplorer is driving all of the interfaces in the above diagram.

Xtensa Xplorer has user interface components for driving trace capture, and also for analyzing trace information – whether it was captured via Xplorer, or captured using the command line tools. On the control side, Xplorer provides for symbolic trace trigger specification, persists your settings (in a launch) and gives convenient actions to drive the *`xt-traxcmd`* tool.

On the analysis side, Xplorer parses the trace information and constructs a call graph which you can navigate using convenient UI actions. It reconstructs addresses into functions based on the Xtensa binary, and lets you navigate the trace in two different disassembly views: one function-oriented; the other execution sequence oriented. As you navigate the trace, it automatically displays the matching source if it can be located.

5.4 Analyzing Performance

After you compile and run the program on the simulator without any errors, the next step is to optimize the performance of the program, as shown in Figure 8. By identifying sections of your program that consume significant CPU cycles, you can optimize those routines.

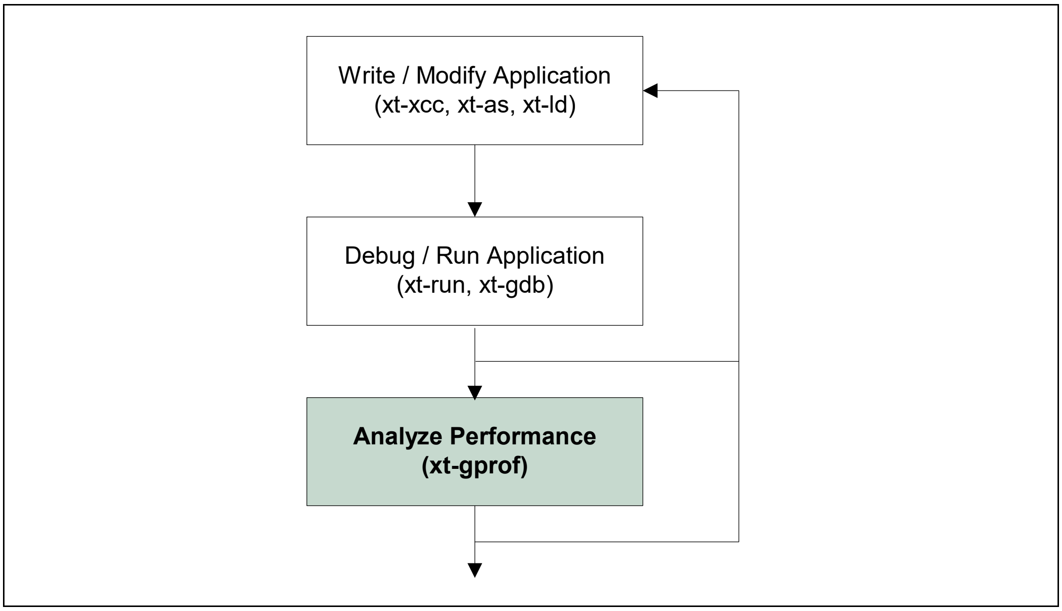


Figure 8. Profiling the Application

Once the “hot spots” of the program have been identified and optimized at the source code level, the performance can often be further improved by adding custom designer-defined instructions specified using the Tensilica Instruction Extension (TIE) language. You can dynamically configure the various software tools to use different sets of designer-defined instructions. See the *Tensilica Instruction Extension (TIE) Language Reference Manual* and the *Tensilica Instruction Extension (TIE) Language User’s Guide* for details.

Several levels of profiling are available. With the `--mem_model` or the `--summary` option, the ISS displays a performance summary of its run (see Section 5.1 on page 19). The simulator can also generate detailed profile information when it is run with the `--profile` option. Xplorer and `xt-gprof` both read that information and display it in various ways, including both flat and hierarchical profiles. Xplorer can also provide graphical views of your programs performance.

Analyzing your program's performance requires two steps: gathering profile information and displaying that data. If you use Xplorer as an IDE, then it handles all this automatically. See the Xtensa Xplorer documentation for more information. This section describes analyzing a program's performance when it has been built and run through Xtensa's command-line tools.

5.4.1 Collecting Profile Data with the Instruction Set Simulator

To collect profiling data, run the simulator as described below. The program does not need to be compiled in any special way, although the output will be more understandable if you use -g when compiling it. If you are using actual hardware, see Chapter 6, which describes hardware profiling.

```
xt-run --profile=main.gmon main.exe
```

The profile client in the ISS can profile certain events instead of just counting processor cycles. For example, you can profile the branch overhead to see where time is spent taking branches in your program. Other supported events include cache misses and pipeline interlocks. For more details on the profile client, see the *Xtensa Instruction Set Simulator (ISS) User's Guide.*

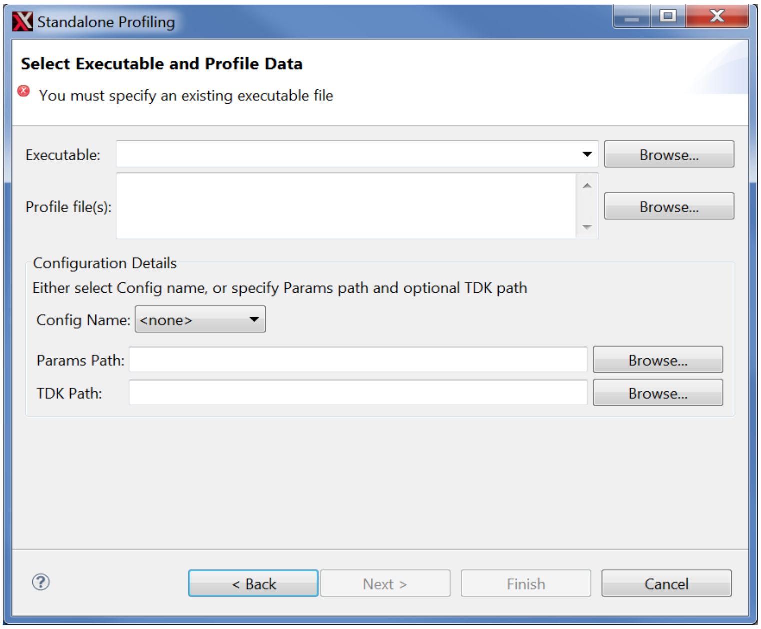
After gathering the simulation data, whether gathered through ISS as described above, or through hardware profiling as described in Chapter 6, you can use either xt-gprof or Xplorer to analyze the data.

5.4.2 Analyzing the Profiling Data with Xplorer

Xplorer generates a useful graphical report with the profiling data. To start Xplorer in standalone profiling mode, type:

```
xplorer --profile <a.out> [<gmon files>..] [-- -data <workspace>]
```

When Xtensa Xplorer starts up in standalone profiling mode as shown in Figure 9, the Select Executable and Profile Data dialog box opens, allowing you to specify the executable and profile files. Ensure that the path to the executable, gmon files and params file are correct, then click Finish to display the profile information.



Select Executable and Profile Data

You must specify an existing executable file

Configuration Details

Either select Config name, or specify Params path and optional TDK path

Config Name: <none>

Params Path:

TDK Path:

Figure 9. Xplorer Standalone Profiling Mode Dialog Box

The target program does not need to have been compiled or executed in the graphical environment, but should have been compiled with `-g` to produce C-source-line information. If no workspace is specified (via the `-data` *`workspace`* argument), Xplorer creates a temporary directory for the workspace. If a workspace is specified, it must not be a sibling or subordinate of any of the source directories.

5.4.3 Analyzing the Profiling Data with xt-gprof

Cadence also provides `xt-gprof` for analyzing the performance data. Invoking it is simple:

```
xt-gprof <executable_name> <profiling_file_name>
```

For example, a session that does compilation, simulation and profiling might look like the following:

```
% xt-xcc -O -g ex1.c -o ex1
% xt-run --profile=gmon.out ex1
% xt-gprof ex1 gmon.out

Flat profile:
                                              self      total
     cumulative        self                 cycles     cycles
  %       cycles      cycles    calls        /call      /call  name

96.89 300006.00  300006.00        1 300006.00  300006.00  init_array
 0.49 301532.00    1526.00       20      76.30      76.30  __udivsi3
 0.45 302928.00    1396.00       20      69.80      69.80  __umodsi3
 0.43 304258.00    1330.00        1    1330.00    7667.42  _vfprintf_r
 0.31 305210.00     952.00                                 ResetH
 0.18 305775.00     565.00        5     113.00     457.20  __udivdi3
 0.18 306320.00     545.00        5     109.00     453.20  __umoddi3
 0.17 306840.00     520.00       20      26.00      26.00  __mulsi3
 0.13 307231.00     391.00       30      13.03      13.03  _mbtowc_r
 0.09 307505.00     274.00                                 _start
 0.08 307761.00     256.00        2     128.00     366.50  __sfvwrite
 0.07 307966.00     205.00        3      68.33      75.72  memchr
 0.06 308156.00     190.00        1     190.00     319.17  _malloc_r
 0.05 308323.00     167.00        3      55.67      55.67  memmove
 0.05 308467.00    144.00       12      12.00      12.00 _WindowUnderflow8
 0.04 308600.00    133.00       12      11.08      11.08 _WindowOverflow8
 0.04 308722.00     122.00        4      30.50      70.83  fflush
 0.03 308804.00      82.00        1      82.00      99.58  __sinit
 0.02 308874.00      70.00        1      70.00     459.33  __smakebuf
 0.02 308929.00      55.00        1      55.00     402.00  exit
 0.02 308982.00      53.00        1      53.00     265.50  _fwalk
 0.02 309032.00      50.00        2      25.00      30.54  sbrk
 0.01 309077.00      45.00        2      22.50     401.00  __sprint
 0.01 309120.00      43.00        1      43.00     514.33  __swsetup
 0.01 309158.00      38.00        1      38.00    7817.00  vfprintf
 0.01 309196.00      38.00                                 _Reset_epilog
 0.01 309233.00      37.00        2      18.50      54.58  _sbrk_r
 0.01 309270.00      37.00        1      37.00    7866.00  printf
```

For more details about the profiler, see the *GNU Profiler User's Guide*

5.4.4 Analyzing the Profiling Data with xt-dgmon

The `xt-gprof` tool shows the user profiling information that is aggregated for functions or source lines. Cadence also provides the `xt-dgmon` command line utility that can be used together with `xt-objdump` or `xt-nm` to correlate profile data to binary files.

When `xt-dgmon is` used together with `xt-objdump`, the profile data is displayed next to each line of disassembly. This is the default mode for `xt-dgmon`.

```
xt-objdump -D | xt-dgmon [--csv] [-[fcp]] \
  <profiling_file_name> [<profiling_file_name> ...]
```

When `xt-dgmon is` used together with `xt-nm`, the profile data is aggregated and displayed for the address range of each symbol corresponding to the executed code. In this mode, the first two whitespace-delimited hex numbers that are outputted by `xt-nm` are interpreted as an address and size. Profiling data is aggregated and displayed for this address range along with the original text.

```
xt-nm -S | xt-dgmon --nm [--csv] [-[fcp]] \
  <profiling_file_name> [<profiling_file_name> ...]
```

The `xt-dgmon` tool supports two output modes:

1. Natural annotation of the input stream. Cycles are displayed as an additional column. Other metrics are displayed on additional lines that follow. This is the default mode.
2. Comma separated values format. All profiling metrics along with the input line are displayed in comma separated values format. This mode is enabled by the `--csv` switch.

By default, “cycles” profiling data is displayed as the first column, regardless of the type of profiling data file that is specified first, and other events are displayed as additional lines (or additional columns) in the same order as specified in the command line. This behavior can be changed using the `-p` flag, which disables any columns from being reordered and forces columns to be in the same order as the profiling data files in the command line.

Filtering of the output is enabled by the `-f` flag: only lines that have profiling data are displayed.

With the `-c` flag, `xt-dgmon` reports how many times an address was a call target. This data is displayed in an additional “calls” column.

The `xt-dgmon` tool aggregates profiling data if multiple data files for the same profiling metric are provided on the command line, but it does not distinguish inclusive profiling from regular, non-inclusive data. To avoid incorrect aggregation, use all inclusive or all non-inclusive data in a given invocation.

For example, the compilation and simulation may look like the following:

```
% xt-xcc -g -O2 hello_world.c
% xt-run --client_cmds="profile --all gmon.out" a.out
```

Next, getting the summary for each function may look like the following (only a portion of the output is shown here):

```
% xt-nm -S a.out | xt-dgmon --nm -fc gmon.out.cyc \
    gmon.out.interlock gmon.out.bdelay

   calls   cycles
       3       33 40006588 00000011 T fclose
                  |-- 12 bdelays
       3      157 40006508 00000080 T _fclose_r
                  |-- 12 interlocks
                  |-- 51 bdelays
       4       76 40003ab0 00000028 T _fflush_r
                  |-- 8 interlocks
                  |-- 24 bdelays
```

In the example above, the function `_fflush_r` was called 4 times, and took 76 cycles to execute, which includes 8 cycles due to interlocks and 24 cycles due to branch delays. Alternatively, getting the same summary in a comma separated values format, it may look like the following (only a portion of the output is shown here):

```
% xt-nm -S a.out | xt-dgmon --nm --csv -fc gmon.out.cyc \
    gmon.out.interlock gmon.out.bdelay

address,size,calls,cycles,interlocks,bdelays,text
0x40006588,0x11,3,33,0,12,"40006588 00000011 T fclose"
0x40006508,0x80,3,157,12,51,"40006508 00000080 T _fclose_r"
0x40003ab0,0x28,4,76,8,24,"40003ab0 00000028 T _fflush_r"
```

Annotation of `xt-objdump` disassembly displays profiling data for each address (only `_fflush_r` code is shown):

```
% xt-objdump -D a.out | xt-dgmon -fc gmon.out.cyc \
    gmon.out.interlock gmon.out.bdelay

   calls   cycles
                         ...

                  40003ab0 <_fflush_r>:
       4       12 40003ab0:     004136          entry   a1, 32
                  |-- 8 bdelays
                4 40003ab3:     228c            beqz.n  a2, 40003ab9
                4 40003ab5:     e288            l32i.n  a8, a2, 56
                8 40003ab7:     289c            beqz.n  a8, 40003acd
                  |-- 4 interlocks
```

```
     40003ab9 <_fflush_r+0x9>:
   4 40003ab9:     069392          l16si   a9, a3, 12
   8 40003abc:     29cc            bnez.n  a9, 40003ac2
     |-- 4 interlocks
            ...

     40003ac2 <_fflush_r+0x12>:
  12 40003ac2:     03bd            mov.n   a11, a3
     |-- 8 bdelays
   4 40003ac4:     02ad            mov.n   a10, a2
   4 40003ac6:     ffeb25          call8   40004978
  12 40003ac9:     0a2d            mov.n   a2, a10
     |-- 8 bdelays
   4 40003acb:     f01d            retw.n
```

Execute the `xt-dgmon --help` command to get a list of options and usage examples.

5.5 Optimizing Code Placement

Code layout in memory may have a significant effect on application performance running on a processor with an instruction cache. The `xt-link-order` tool allows you to optimize instruction cache performance by ordering code sections based on the profile data. The final reordering is performed by the linker.

Follow these steps to optimize code section placement using `xt-link-order`:

1. Use the `-ffunction-sections` XCC command-line option to compile your application, and the `--xt-map` linker option (`-Wl,--xt-map` if linking with XCC) to link the executable.

   ```
   xt-xcc -ffunction-sections -c -o example.o example.c
   xt-xcc -Wl,--xt-map example.o
   ```

2. Run the application to produce one or more profile data (gmon) files, as described in Section 5.4.
3. Invoke the xt-link-order tool with the original executable and profile data files as inputs to produce a section ordering specification file as the output:

   ```
   xt-link-order -e a.out -g gmon.out.cyc -o section_order.txt
   ```

4. Use the `--sections-placement` linker option (`-Wl,--sections-placement` if linking with XCC) to relink the application with the section ordering generated in step 3:
   ```
   xt-xcc example.o -Wl,--sections-placement=section_order.txt
   ```

The `xt-link-order` tool can also use profile data for placing frequently used code into IRAM. To enable this feature pass `--use-iram` command-line option on the step 3 above. The tool determines free space in the IRAM by analyzing the executable and comparing its content to the Xtensa core configuration. If there is a free space available, `xt-link-order` selects which sections to place into IRAM. You can also override the Xtensa core configuration parameters and instruct the tool to use less than the total available IRAM size by specifying one or more options:

```
--iram=SECTION:[LITERAL_SECTION:]SIZE
```

where *`SECTION`* is the IRAM output section name, *`LITERAL_SECTION`* is the section name for literals if different from *`SECTION`*, and *`SIZE`* is the IRAM size in bytes. For example: `--iram=.iram0.text:1024` will instruct the tool to put up to 1024 bytes of code and literals into `.iram0.text` section.

Any legal gmon profile file can be used, but best results are usually achieved with either cycle profiling (gmon.out.cyc) or instruction cache miss profiling (gmon.out.icmiss). When using IRAM placement, gmon.out.cyc will often maximize the percentage of instructions that are fetched from IRAM, potentially improving power. Using gmon.out.icmiss will often result in the fewest instruction cache misses, improving performance.

The sections order output is a text file that instructs the linker to place certain input sections into the IRAM and specifies the relative order of the input sections. The IRAM placement command is a single line with the following syntax:

```
put: section (input_section|function_name) [file_pattern]
```

where `section` is the target section name; `input_section` is the section name in an object file that the linker should place into `section`, alternatively a function name can be used; `file_pattern` is an optional object file name pattern.

Sections order is defined by a number of lines with the following syntax each:

```
(input_section|function_name) [file_pattern]
```

The linker orders input sections in the executable in the same order as they are listed in the file.

Please note that the function name in the patterns above can be used to match only functions compiled with `-ffunction-sections` option. Function name matches both code and literal sections that correspond to the function.

The object file name pattern allows one to specify the section source file. It may contain special characters: ’*’ (match any number of any characters) and ’?’ (match any single character). Please see the linker-generated map file if you are not sure which object file paths the linker actually uses.

The following is an example of an ordering file.

```
#
# IRAM placement
#
put: .iram0.text    .literal._Z5func1RiS_   helloworld.o
put: .iram0.text    .text._Z5func1RiS_      helloworld.o
put: .iram0.text    _Z5func2RiS_            helloworld.o
put: .iram0.text    func3                   helloworld.o
#
# sections order
#
.text._Z5func4RiS_      helloworld.o
_Z5func5RiS_
func6                   helloworld.o
func7                   path/*/example.o
.text                   example.o
.text                   path/libname.a:example.o
```

Additional options for the section reordering tool are described in its online documentation available via

```
xt-link-order --help.
```

5.6 Static Stack Usage Analysis

The Xtensa tool set provides the `xt-stack-usage` command line utility for estimating maximum amount of stack space an executable or object file might use. In addition to maximum amount the utility reports stack frame information for individual functions, estimated stack usage by a function with all its callees, reports encountered problems which may affect estimate.

The input file for the tool can be an executable or object file. It doesn't work with static library files (archives). The input file must have debug information included, please use `-g` XCC option and don't strip symbols. The tool analyzes binary code and debug information to detect stack space that the program may use.

`xt-stack-usage` prints a report that consists of the following sections:

- Report header which includes stack usage estimate
- Functions list with stack usage information for each function
- Recursion cycles report lists identified recursions
- If it's an object file undefined references list printed with a list of functions that refer each undefined symbol

- Legend - columns and attributes short description

Example of report header:

```
Stack usage: 368+ bytes.

This estimate is a lower bound. Stack usage might be higher for the fol-
lowing reasons:

* detected functions with variable stack frame size ('D' function attri-
bute)

Call chain:
         Inclusive
Depth   Size  Attrs    Size   Attrs  Name
[ 1]     368+ D-----     32  ------- main
[ 2]     336+ D-----     48+ D------ bar
[ 3]     288  ------     32  ------- foo
[ 4]     256  ------     80  ------- sqrt
[ 5]     176  ------     96  ------- __ieee754_sqrt
[ 6]      80  ------     80  ------- __propagateFloat64NaN

Entry points:
         Inclusive
Index   Size  Attrs    Size   Attrs  Name
[ 1]     368+ D-----     32  ------- main
```

The `'+'` suffix in the stack usage estimate indicates that the actual value might be greater and stack usage may reach 368 bytes or higher. This estimate is followed by a list of reasons why it's not exact. The full list of reasons is:

- Detected function with variable stack frame size. This happens if a function modifies the stack pointer in an unusual way or if it uses the `alloca` function. Absence of debug information might cause this inaccuracy too.
- Could not find indirect call target. The tool could not identify which function is called. Usually this happens if a function is called using pointer to this function, for example if it's a callback or an entry in a functions table. C++ virtual methods may cause this too.
- Could not find indirect jump targets. The tool could not identify jump targets of the `jx` instruction. In a regular C code this might be caused by a lack of debug information.
- Calls to undefined symbols detected or jumps to undefined symbols detected. These are triggered by external references in object files. The undefined references report lists which symbol is referenced from which function.

- Recursion detected. Any recursion makes stack usage estimate inexact. The recursions report provides more information on which functions are part of recursion cycles.

Next part of report is a call chain that corresponds to the reported stack usage amount.

Note: If an estimate is not exact, other call chains may have higher stack usage than the reported one. If an estimate is exact, there are might be other call chains that have same stack usage but only one is reported.

Finally, each entry point is listed. The designer can specify entry points using `--entry=SYMBOL` command line option. In the absence of `--entry` options the `main` symbol is used.

In all reports, the function information line has the following columns:

Table 6. Function Information Line Columns

Column	Description
Inclusive Size	maximum stack usage after taking into account all reachable callees; value is followed by `'+'` if the size might be greater than displayed size;
Inclusive Attrs	attributes "Dcjuxr"; attribute is set if the function or any reachable callee has this attribute;
Size	stack frame size in bytes; value is followed by `'+'` if the size might be greater than displayed size;
Attrs	individual function attributes "Dcjuxda";
Align	stack frame alignment for functions that align stack pointer; `'*'` if function doesn't align stack pointer (optional column);
Address	function address in <section_index>:<address> format (optional column); section index corresponds to the output of `xt-readelf -S` command;
Name	function name and optionally source file and line number;

Table 7. Attributes Description

Attribute	Description
D	function has dynamic frame size or modifies stack pointer;
c	could not find call target (indirect call);
j	could not find jump target (indirect jump);
u	function calls undefined symbol;
x	function jumps to undefined symbol;

Table 7. Attributes Description

Attribute	Description
d	function is not reachable from entry points; this might be a dead code, a function reachable from unaccounted entry point or it's the tool could not find caller of this function due to unresolved indirect calls or jumps; not used in inclusive attributes;
a	function aligns stack pointer, actual stack usage might be higher than shown in **Size** column due to aligning; this attribute doesn't lead to `'+'` suffix in stack frame size; not used in inclusive attributes;
r	function belongs to recursion loop or there is a call path from this function that leads to recursion loop; used in inclusive attributes only;

The functions list report displays information for each function:

```
   Inclusive
  Size  Attrs    Size   Attrs   Name

   432+ D-----     64  ------- main
   368+ D-----     64+ D------ bar
   304  ------     48  ------- foo
   256  ------     80  ------- sqrt
   176  ------     96  ------- __ieee754_sqrt
    80  ------     80  ------- __propagateFloat64NaN
    64  ------     32  ------- matherr
    64  ------     64  ------- __fpclassifyd
    48  ------     48  ------- __muldf3
```

During stack usage estimation, all recursion cycles in the static call graph are ignored. This makes estimation inaccurate. The tool identifies clusters of functions that call each other and prints recursions report with a call cycle for each cluster. In the example below, a cycle from two functions is shown - the function `bar_r` calls function `foo_r` which calls `bar_r` again:

```
     Inclusive
    Size  Attrs    Size   Attrs   Name

Call cycle:
    128+ -----r      64  ------- bar_r
```

```
     64+ -----r      64  ------- foo_r
    128+ -----r      64  ------- bar_r
```

Finally, the undefined references report is shown. Example below is the output for an object file which has external references to `__mulsi3` function, called from `foo` and `bar`, and a reference to `fprintf` function called from `main`:

```
     Inclusive
    Size  Attrs    Size   Attrs  Name

References to symbol __mulsi3:
     128+ D--u--     64+ D--u--- bar
      64+ ---u--     64  ---u--- foo

References to symbol fprintf:
     192+ D--u-r     64  ---u--- main
```

The output of the `xt-stack-usage` tool is configurable. It's possible to show or hide optional columns and turn on or off individual reports. Please use `--help` command line parameter to get a list of available options.

5.7 Source Code Coverage

Starting with the RH-2018.5 release, the Xtensa LLVM C/C++ compiler (`xt-clang`, `xt-clang++`) can be used to instrument the code for collecting source code coverage information. You can do this simply by adding the `--coverage` command-line option at both the compile and link steps. To ensure the accuracy of instrumentation, the compilation should be done without optimizations (`-O0`).

With the `--coverage` option the compiler will generate a `.gcno` data file for each object file. When you simulate the instrumented program, a `.gcda` files will be generated for each object file. Together, these two sets of files contain the source code coverage information. You can then view this information with the `xt-cov` command-line tool, which is based on the `llvm-tool` from the LLVM compiler infrastructure project. Only the `gcov` mode is currently supported:

```
xt-cov gcov [options] sourcefile ...
```

You should invoke `xt-cov` from the same directory where you ran the compiler. For each specified source file, an output file with execution counts is created by adding a `.gcov` suffix. Command-line options for `xt-cov gcov` are described in Table 8.

Table 8. Command-line options for `xt-cov gcov`

Option	Description
`-a`	Display the information for all basic blocks in cases where there are multiple blocks for single source line.
`-b`	Display branch probabilities.
`-c`	Display branch counts instead of probabilities (requires `-b`).
`-f`	Show a summary of coverage for each function.
`-help`	Display available options.
`-n`	Display only coverage summary but do not generate `.gcov` files.
`-o=<`*`dir`*`|`*`file`*`>`	Find object files in the specified directory or based on the specified file's path.
`-u`	Include unconditional branches with `-b`.

6. Hardware-Based Performance Analysis

In addition to the ISS-based tools described in Chapter 5, we provide several methods for analyzing a program run in a hardware environment. In Section 6.1, we describe profiling on hardware targets. The profiler allows one to automatically see, using statistical sampling, how many cycles (or events such as cache misses) are spent in each function or line of the source application. In Section 6.2, we describe the Performance Counter API that allows an application to directly measure events at program specified times. The profiler, for example, might tell you that approximately 20% of your data cache misses happen while executing a particular C function. The Performance Counter API allows you to measure exactly how many data cache misses happen during a particular phase of an algorithm.

6.1 Hardware-Based Profiling

This feature provides similar functionality to the `--profile` option of ISS but uses statistical sampling based on timer or Performance Monitor interrupts. The profiler, implemented as a target library, periodically records the program counter value. At the end of the profiling session, accumulated data is transferred to the host. The debugger connected to the OCD Daemon is used to accomplish the file I/O to and from hardware. Unlike the ISS, hardware-based profiling both perturbs the profiled code and is subject to statistical sampling errors.

Note: File I/O over the debugger connection requires the use of the `libgdbio` library. So it is necessary to use the `gdbio` LSP when linking the executable.

There are two hardware profiling libraries provided. One uses a built-in timer. The API specific to this solution is discussed in Section 6.1.1. The second uses the optional Performance Monitor module which contains up to 8 performance counters. The API specific to this solution is discussed in Section 6.1.2. Finally, Section 6.1.3 covers API functions and usage details that are common to both libraries.

The API for both libraries is prototyped in the `<xtensa/xt_profiling.h>` header file.

6.1.1 Hardware Profiling Based on Timer Interrupt

Hardware profiling based on timer interrupts requires that the Xtensa configuration have a dedicated timer available. For XEA2 based configurations the timer must be assigned to a dedicated interrupt level higher than 1. Only software running at an interrupt level below that of the timer will be profiled. The default sampling interval is 16384 cycles. The interval may be changed by calling:

```
    xt_profile_set_frequency(unsigned int freq)
```

from the target process. The `freq` argument should be a multiple of 1024 to avoid round-off errors.

The timer-based profiling feature also works on the ISS. In this case the library uses the simulator's file I/O support to write out the results file. This is not particularly useful since the ISS has built-in support for profiling, but useful for debugging or sanity checking the results obtained from hardware.

In addition to XTOS, the timer-based profiling feature also works with XOS. The XOS library provides the necessary functions to interface with the library. If used with XOS, the timer must be selected with care so that XOS and the profiler do not end up trying to use the same timer. The Xtensa configuration needs to have at least two timers available.

6.1.2 Hardware Profiling Based on Performance Counters

Hardware profiling based on performance counters provided with the Performance Monitor uses a special interrupt of type *Profiling*. The Profiling interrupt can be assigned by the user to a specific interrupt level. The implementation of the software interface to the interrupts required for performance monitoring is provided in XTOS. The software interface required to work with XOS is also provided. For any other RTOS, the interface functions should be implemented following the XOS version as a guide.

The Performance Monitor can use hardware counters to monitor various events. The profiler samples each configured performance counter with a specific interval and produces a separate profiling file for each counter. This allows the monitor to sample not only cycles but also events like cache misses, branch delays and many other. See Section 8.5 "Counter Overview" in the " *Debug Guide*" for more details on the types of events that can be counted. You can select which events to count either using the launch menu in the Xplorer IDE or using the following API functions directly in the target code.

The following function stops profiling if it was enabled and clears all hardware performance counters:

```
    void xt_profile_config_clear();
```

If profiling had previously been enabled, all previously accumulated profile data is retained and will be saved to a separate `gmon` files at program exit or during the next call to `xt_profile_save_and_reset()`.

You can get the number of free (not yet configured) hardware performance counters using this function:

```
    int xt_profile_config_num();
```

The following function allocates and configures hardware performance counter for profiling:

```
int xt_profile_config_counter(unsigned int selector,
                              unsigned int mask,
                              unsigned int trace_level,
                              unsigned int period);
```

- `selector` is the event group (one of `XTPERF_CNT_*` values defined in `<xtensa/xt_perf_consts.h>`);
- `mask` is the subset within the group (a combination of `XTPERF_MASK_*` values defined in `<xtensa/xt_perf_consts.h>` for `selector`);
- `trace_level` specifies interrupt levels at which to take samples; if `trace_level` is greater than or equal to zero, samples are taken only at interrupt levels below or equal to `trace_level`; if `trace_level` is negative samples are taken only at (`-trace_level`) interrupt level or higher;
- `period` is the non-zero sampling interval.

Returns zero on success or non zero error code if failed. The following error codes are defined:
`XTPROF_ERR_OUT_OF_MEM` - failed to allocate memory;
`XTPROF_ERR_INVALID_ARGS` - invalid function arguments;
`XTPROF_ERR_NOT_ENOUGH_COUNTERS` - no more free performance counters available.
`XTPROF_ERR_DEFUNCT` - profiling is disabled because of previous errors (typically because of out of memory errors during profiling);

In addition to the selectors and masks covered in Table 8-39 of the " *Debug Guide*" `<xtensa/xt_perf_consts.h>` header file defines the high level events shown in Table 9. These events are composite events created by appropriately combining multiple hardware counters. They are designed to more closely match the events counted by the ISS.

Table 9. High Level Events

Slector	Mask	Description
XTPERF_CNT_COMMITTED_INSN	XTPERF_MASK_COMMITTED_INSN	Committed instructions
XTPERF_CNT_BRANCH_PENALTY	XTPERF_MASK_BRANCH_PENALTY	Branch penalty cycles
XTPERF_CNT_PIPELINE_INTERLOCKS	XTPERF_MASK_PIPELINE_INTERLOCKS	Pipeline interlocks cycles
XTPERF_CNT_ICACHE_MISSES	XTPERF_MASK_ICACHE_MISSES	ICache misses penalty in cycles
XTPERF_CNT_DCACHE_MISSES	XTPERF_MASK_DCACHE_MISSES	DCache misses penalty in cycles

The following function controls the sampling period randomization option:

```
void xt_profile_randomization(int value);
```

Without randomization, the profiler uses a constant sampling period. This is the default mode. In cases when the code is highly periodic, you may see aliasing effects with events attributed to one line of code more often than to another. This happens because of one specific sampling period value and not because of an actual difference in the event distribution. For example, in a loop all the cycles might be attributed to one particular instruction even though the actual cycles are distributed among all the instructions in a loop. To reduce such aliasing effects, you can enable sampling period randomization. Here individual sampling intervals will be chosen semi-randomly with an average sampling interval given by the period specified at performance counter initialization.

- `value` disables randomization if 0, enables if non zero.

The following example configures sampling of two events: branch penalty performance and a more specific counter for stalls due to instruction memory accesses, and starts profiling:

```
#include <xtensa/xt_profiling.h>
...
// clear previously configured hardware counters
xt_profile_config_clear();
// profile branch penalty cycles
xt_profile_config_counter(XTPERF_CNT_BRANCH_PENALTY,
  XTPERF_MASK_BRANCH_PENALTY, 0, 1024);
// profile uncached instruction fetch stalls
xt_profile_config_counter(XTPERF_CNT_I_STALL,
  XTPERF_MASK_I_STALL_UNCACHED_FETCH, 0, 4096);
// start profiling
xt_profile_enable();
```

Note: There is an overhead associated with events recording. Choosing too small an interval will lead to inaccuracies due to the perturbation overheads of the library, while choosing too large an interval may result in statistically too few samples to derive accurate average numbers of a specific event in an application.

The performance monitoring library can measure events even from code inside of higher priority interrupts handlers. However, the recording of events is postponed until the return from a higher priority interrupt routine. If there are two or more performance monitoring overflows, either from the same or different events, before the high priority interrupt routine exits, all but the first event will be dropped. This effect can be minimized by using a sufficiently large interval, by making sure that the program does not spend too much time in higher priority interrupts or by selecting the appropriate interrupt level for the Profiling Interrupt.

By default, the performance monitoring library uses interrupt management routines provided by XTOS. For integration with other runtime libraries or RTOS, the following interrupt management functions must be provided:

```
void xt_profile_set_interrupt_handler(unsigned int interrupt, void
*handler);
void xt_profile_set_timer_handler(unsigned int timer, void *handler);
void xt_profile_interrupt_on(unsigned int interrupt);
void xt_profile_interrupt_off(unsigned int interrupt);
void xt_profile_set_timer(unsigned int timer, unsigned int cycles);
```

Note: These routines are already provided for integration with XOS.

Note: These functions can be called before entering or after exiting from `main`.

6.1.3 Common API and other topics

To use hardware profiling outside of the Xplorer IDE, first (optionally) compile the code with `-hwpg[=`*n*`]`, then link the code using XCC with `-hwpg[=`*n*`]` flag. Then run the program on the hardware as you normally would to debug it.

Both, hardware profiling using timers and hardware profiling using performance counters, can provide two different levels of detail.

- If the code is linked, but not compiled, using `-hwpg=`*n* (where *n* is the timer number allocated for profiling: 0, 1, or 2) or `-hwpg` (for profiling based on the Performance Monitor option), the samples taken for a given function are recorded, but the number of calls to each function is not counted. This method minimally perturbs the code and does not impact the size of the program, but provides less detail than the second method, below. In particular, the profiler can estimate the number of events spent directly in a particular function but can not attribute to a function the events that occur while that function is calling another function.
- If the code is both compiled and linked using `-hwpg=`*n* or `-hwpg`, both a given function's samples taken and the number of times it is called is counted. This method perturbs the code by calling a library instrumentation function on entry to every target function, but provides more detailed performance information, like the function call graph.

If your program calls `exit()`, just quit from the debugger when your program finishes. It will create a file named `gmon.out.000000` in the current directory (if a file of that name exists, it tries `gmon.out.a00000`, *etc.*, until it finds a name that does not exist). If your program does not call `exit()`, you must call the function `xt_profile_save_and_reset()` either in the application or interactively in the debugger. To call it interactively in `xt-gdb`, interrupt the program (use control-C), then type `call xt_profile_save_and_reset()`.

Profiling can be turned off and on programmatically at any point by calling the following functions:

```
xt_profile_disable()
xt_profile_enable()
```

These functions can be used if there are portions of your code you wish not to profile. Whether profiling is enabled at startup or not depends on the profiling library you use and your environment.

The timer-based profiler is always enabled at startup. If your environment supports `.init` sections (the default environment provided by does), then hardware profiling is automatically turned on, just prior to entering your application's "`main`" function. If you wish to avoid profiling your application's initialization, then turn off profiling early in `main`, and turn it back on at an appropriate time. If your environment does not support `.init` sections, then you will need to turn profiling on with the function above at an appropriate time.

The profiler based on Performance Monitor is turned on at startup only if it runs in the Xplorer IDE and only if your environment supports `.init` sections. If it's not the case you need to configure the performance counters and enable profiling in your target code as shown in the Section 6.1.2.

Hardware-based profiling requires memory. Ordinarily, it allocates this memory during its initialization through a call to `sbrk()`, which is a low-level system call that allocates memory from the default heap. (The C-library function `malloc` normally uses `sbrk` to get large blocks of memory, which it then subdivides, so the hardware-profiler does not depend on `malloc`, but on a lower-level system call.) If a call to `sbrk()` fails, the library prints an error via the debugger at the end of profiling which says "Hardware Profiling Error: couldn't allocate memory."

To have the hardware profiler allocate its memory from a different location, write your own function with the prototype, "`void * xt_dbfs_sbrk(int bytes);`", which returns a `bytes`-sized block of memory, aligned to 4 bytes, or 0xffffffff if insufficient memory is available. This function should be placed in its own separate file and this file should not be compiled with the `-hwpg` option.

The amount of memory allocated during initialization is approximately the same as the executable code size as reported by the `xt-size` tool in the 'text' column. Under certain conditions, the profiler may need more memory. In such cases, the library prints an error via the debugger at the end of profiling which says "Hardware Profiling Error: ran out of memory.". No profile data is saved. If this happens you can provide the library with more memory by calling `xt_profile_add_memory` early in the application.

```
xt_profile_add_memory(void * buf, unsigned int buf_size)
```

Please note that the buffer should be zero initialized before calling this function.

When profiling is complete, the data is transferred from the target process across a relatively slow uplink (8k bytes/second for XOCD). Thus, it may take some time for the file to be written to the host file system.

You may examine the generated `gmon` files using `xt-gprof` or the Standalone Profiler (see Section 5.4.2 “Analyzing the Profiling Data with Xplorer”).

6.2 Performance Counters API

Xtensa processors with the Performance Monitor option provides a set of configurable performance counters. As described in Section 6.1, these counters are used to generate profiling information. The performance counters can also be accessed from your application at run time allowing you to directly measure how many events (for example, cache misses) occurred during a particular phase of your program.

Note: This feature is not available in the ISS, or in XTMP or XTSC system simulations.
Note: This feature and the hardware profiling feature described in the Section 6.1 above are not compatible and cannot be used simultaneously. Attempt to profile application that uses performance counters API will result in run time error with message: "linked with libperfmon"

To access the performance monitoring library from your program, you should include the `<xtensa/xt_perfmon.h>` header file and link with the `-lperfmon` option.

The following function stops events counting and clears all hardware performance counters:

```
void xt_perf_clear();
```

You can get the number of free (not yet configured) hardware performance counters by calling:

```
int xt_perf_counters_num();
```

The following functions allocate and configure performance counter:

```
counter_id_t xt_perf_init_counter32(unsigned int selector,
                                    unsigned int mask,
                                    unsigned int trace_level);

counter_id_t xt_perf_init_counter64(unsigned int selector,
                                    unsigned int mask,
                                    unsigned int trace_level);
```

Hardware counters are 32-bit, but the library can emulate 64-bit counters by handling a special interrupt of type *Profiling* which is triggered each time a 32-bit hardware counter overflows. Please note: this is different from using two or more hardware performance counters as a single wide counter described in the " *Debug Guide*". This function disables counting if it was enabled.

- `selector` is the event group (one of `XTPERF_CNT_*` values defined in `<xtensa/xt_perf_consts.h>`);
- `mask` is the subset within the group (a combination of `XTPERF_MASK_*` values defined in `<xtensa/xt_perf_consts.h>` for `selector`);
- `trace_level` specifies interrupt levels at which to events counted; if `trace_level` is greater or equal to zero events are counted only at interrupt levels below or equal to `trace_level`; if `trace_level` is negative events are counted only at `(-trace_level)` interrupt level or higher.

The function returns a negative value if it fails. This indicates that the function arguments are invalid or all hardware performance counters are already configured. If the function call is successful, it returns the counter id. This can be used later to access the counter values. Note that the counter id is valid until the call to `xt_perf_clear()`.

You can turn counting on and off using following functions:

```
void xt_perf_enable()
```

```
void xt_perf_disable()
```

Counting is disabled at program start before `main` is executed. These functions preserve counters configuration and values.

A 32-bit counter may overflow. If this happens its value wraps around and internal overflow flag is set for this counter. You can read the overflow state using the following function:

```
int xt_perf_overflow32(counter_id_t counter_id);
```

Returns a negative value if the counter id is not valid, zero if counter has not overflowed, positive if it has overflowed.

The following function resets counter value to zero and drops overflow flag if it was set:

```
int xt_perf_reset_counter(counter_id_t counter_id);
```

It works with both 32- and 64-bit counters. Returns non-zero on success or zero if counter id is not valid.

The following two functions allow you to read the current counter value:

```
unsigned int xt_perf_counter32(counter_id_t counter_id);
```

This function returns current 32-bit counter value for counter id allocated using `xt_perf_init_counter32` function. Returns zero if counter id is not valid.

```
unsigned long long xt_perf_counter64(counter_id_t counter_id);
```

This function returns the current 64-bit counter value for counter id allocated using `xt_perf_init_counter64` function. Returns zero if the counter id is not valid.

The following example shows how to configure two counters and read their value:

```
#include <stdio.h>
#include <xtensa/xt_perfmon.h>

void foo(int i) {
  if (i>0)
    foo (i-1);
}

int main (int argc, char *argv[]) {
  counter_id_t c1, c2;
  unsigned int v1, v2;
  // clear counters
  xt_perf_clear ();
  // count cycles in c1 counter,
  // this doesn’t include window overflow/underflow
  // exceptions as trace level is set to zero
  c1 = xt_perf_init_counter32 (XTPERF_CNT_CYCLES, XTPER-
F_MASK_CYCLES, 0);
  // count exceptions and pipeline replays in c2
  c2 = xt_perf_init_counter32 (XTPERF_CNT_EXR, XTPERF_MASK_EX-
R_ALL, 0);
  // start counting
  xt_perf_enable ();
  // this function may generate window overflow
  // exceptions on windowed ABI
  foo (32);
  // disable counting
  xt_perf_disable ();
  // reading counters value
  v1 = xt_perf_counter32 (c1);
  v2 = xt_perf_counter32 (c2);
  printf ("Cycles: %u\n", v1);
  printf ("Exceptions and replays: %u\n", v2);
```

```
    return 0;
  }
```

By default, the performance counter library uses interrupt management routines provided by XTOS. For integration with XOS or another RTOS, the following interrupt management functions must be provided:

```
void * xt_perf_set_interrupt_handler(uint32_t interrupt, void *
handler);
void xt_perf_interrupt_on(uint32_t interrupt);
void xt_perf_interrupt_off(uint32_t interrupt);
```

Providing these functions in your application will override the weak versions of the functions supplied in the library. For reference, the following implementations can be used for XOS:

```
#include <xtensa/xos.h>

void *
xt_perf_set_interrupt_handler(uint32_t interrupt, void * handler)
{
     xos_register_interrupt_handler(    interrupt,
                                  (XosIntFunc *) handler,
                                  XOS_NULL);
     return NULL;
}

void
xt_perf_interrupt_on(uint32_t interrupt)
{
     xos_interrupt_enable(interrupt);
}

void
xt_perf_interrupt_off(uint32_t interrupt)
{
     xos_interrupt_disable(interrupt);
}
```

6.3 Hardware-Based Feedback

The toolchain allows you to use XCC's feedback-directed compilation in a hardware environment without a file system. Like hardware profiling, hardware feedback uses the debugger to accomplish the file I/O. Note that code compiled for feedback is approximately 4 to 5 times larger and slower than normal. Please verify that your Xtensa configuration and memory layout can accommodate the increased memory requirements before attempting to use this feature.

To use hardware feedback in Xplorer, select the `hardware feedback` option in the `Optimization` tab of the `Xtensa Project Build Properties` window.

Outside of Xplorer, first compile and link your program using `-fb_create_HW` *`prefix`*.

Then run the program on the hardware as you normally would to debug it, either through `xt-gdb` or `xplorer --debug` *`program`*. Your program will be run under the debugger. If it calls `exit()`, just quit from the debugger when your program finishes. It will create a file named *`prefix`*`.000000` (if a file of that name exists, it tries *`prefix`*`.a00000`, etc., until it finds a name that does not exist). If your program does not call `exit()`, you must call the function `xt_feedback_save_and_reset()` either in the application or interactively in the debugger. See Section 6.1 for instructions on how to call a function interactively.

Now, recompile your program using `-fb_opt` *`prefix`*. Note: You may generate multiple feedback files with a given *`prefix`* before recompiling with `-fb_opt`, and they will be averaged together by the compiler, weighted by their execution time. See the *Xtensa C and C++ Compiler User's Guide* for details.

Feedback accumulation can be turned off and on programmatically at any point by calling the following functions prototyped in the `<xtensa/feedback.h>`.

```
xt_feedback_disable()

xt_feedback_enable()
```

Hardware-based feedback requires memory which it ordinarily allocates through a call to "sbrk", which allocates memory from the heap. To change this behavior, write your own function with the prototype, `"void * xt_dbfs_sbrk(int bytes);"`. This function should follow `sbrk` semantics. Feedback will automatically use your function.

Index