ThreadArch Toolchain

User Guide and Reference Manual

Version 2.4.6

June 2018

**Disclaimer:**

**Redpine Signals, Inc**.

2107 N. First Street, #540

San Jose, CA 95131.

Tel: (408) 748-3385

Fax: (408) 705-2019

Email: [sales@redpinesignals.com](mailto:sales@redpinesignals.com)

Website: [www.redpinesignals.com](http://www.redpinesignals.com)

The information in this document pertains to information related to Redpine Signals, Inc. products. This information is provided as a service to our customers, and may be used for information purposes only.

Redpine assumes no liabilities or responsibilities for errors or omissions in this document. This document may be changed at any time at Redpine’s sole discretion without any prior notice to anyone. Redpine is not committed to updating this document in the future.

Copyright © 2018 Redpine Signals, Inc. All rights reserved.

**About this Document**

The ThreadArch Toolchain User Guide and Reference Manual describes the toolchain and provides installation instructions as well as instructions on its usage.

The ThreadArch toolchain includes an assembler, linker, librarian, object file dumper, debugger, and simulator.

Table Of Contents

[1 Installation of the Toolchain 7](#_Toc517950211)

[1.1 Introduction 7](#_Toc517950212)

[1.2 Hosts/Targets Supported 7](#_Toc517950213)

[1.3 Minimum System Requirements 7](#_Toc517950214)

[1.4 Installation Procedure 7](#_Toc517950215)

[2 About ThreadArch Toolchain Tools 9](#_Toc517950216)

[2.1 GCC 9](#_Toc517950217)

[2.2 GDB 9](#_Toc517950218)

[2.3 Binary Utilities (Binutils) 10](#_Toc517950219)

[2.4 Toolchain Flow 11](#_Toc517950220)

[3 Compiling and Simulating 12](#_Toc517950221)

[4 Compiling and Running a Program on a Target 14](#_Toc517950222)

[5 Compiling and Running a Multi-Threaded Program 16](#_Toc517950223)

[6 Create a library 18](#_Toc517950224)

[7 Introduction to Optimization 19](#_Toc517950225)

[8 Basic Application Binary Interface Specification 20](#_Toc517950226)

[8.1 Scope 20](#_Toc517950227)

[8.2 Compiler 20](#_Toc517950228)

[8.2.1 Command Line Options 20](#_Toc517950229)

[8.2.2 Data Types and Arguments 20](#_Toc517950230)

[8.2.3 Registers 20](#_Toc517950231)

[8.3 Calling Conventions 21](#_Toc517950232)

[8.3.1 Stack Allocation 21](#_Toc517950233)

[8.3.2 Stack Before Calling Function foo() 23](#_Toc517950234)

[8.3.3 Stack after Calling the Function foo() 23](#_Toc517950235)

[8.3.4 Passing Arguments 24](#_Toc517950236)

[8.3.5 Function Return 24](#_Toc517950237)

[8.4 Assembler 24](#_Toc517950238)

[8.4.1 ThreadArch Assembler Syntax 24](#_Toc517950239)

[8.4.2 Directives 24](#_Toc517950240)

[8.5 Object File Formats 24](#_Toc517950241)

[8.5.1 Sections Information 25](#_Toc517950242)

[8.5.2 Symbol Table Format 25](#_Toc517950243)

[8.6 Debugger 25](#_Toc517950244)

[8.6.1 Connecting to Simulator 25](#_Toc517950245)

[8.6.2 Connecting to Target 25](#_Toc517950246)

[8.6.3 Command Line Options 25](#_Toc517950247)

[8.6.4 Debugging Command 26](#_Toc517950248)

[8.6.5 Compiling the driver Source 26](#_Toc517950249)

Table of Figures

[Figure 1: Toolchain Flow 11](#_Toc517950250)

[Figure 2: Multi threaded program 16](#_Toc517950251)

[Figure 3: Stack allocation before a fucntion call 21](#_Toc517950252)

[Figure 4: Stack allocation after a function call 22](#_Toc517950253)

Table of Tables

[Table 1: GCC Command Line Options 9](#_Toc517950254)

[Table 2: GDB Commands 10](#_Toc517950255)

[Table 3: Binutils Options 11](#_Toc517950256)

[Table 4: Data types 20](#_Toc517950257)

[Table 5: Regsiters 21](#_Toc517950258)

[Table 6: Assembler Syntax 24](#_Toc517950259)

# Installation of the Toolchain

## Introduction

The ThreadArch Toolchain is a collection of tools such as compiler, assembler, linker, librarian, object file dumper, debugger, and simulator, etc. ThreadArch compiler is a cross compiler which generates binary executable files to execute on the ThreadArch processor.

## Hosts/Targets Supported

* ThreadArch Toolchain Toolkit supports the host: Red Hat Enterprise Linux 3.0
* ThreadArch Toolchain Toolkit supports the targets: rpine-ta-elf32

## Minimum System Requirements

|  |  |
| --- | --- |
| Host | x86 Linux PC |
| Host Version | Red Hat Enterprise Linux 3.0 |
| RAM | 32 MB |
| Hard Disk | ThreadArch tools for newlib (46 MB) |

## Installation Procedure

ThreadArch toolchain distribution contains gnu-ta-2.4.5-20130704.tgz. To install ThreadArch Toolchain, follow the steps given below.

Step -1: untar the tgz file using following command:

Tar xvzf gnu-ta-2.4.5-20130704.tgz

Step – 2: Set the environment variables.

In order to use the ThreadArch toolchain, set some environment variables for specifying the path of Toolchain binaries.

* Set the environment variables for installed toolchain. If you have installed the ThreadArch toolchain to a different path, please specify the appropriate path instead of the default path.
* Type the following command on the command prompt if your shell is 'bash' shell.

OR

* Copy the commands to the '.bashrc' file and execute the following command.
* source ~/.bashrc

Environment variables settings for Tool Chain:

export PATH=$PATH:/opt/gnu-ta/bin:/opt/gnu-ta/lib/gcc-lib/rpine-ta-elf32/3.3

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/opt/gnu-ta/lib: /opt/gnu-ta/lib/gdb:/opt/gnu-ta/lib/gcc-lib:/opt/gnu-ta/rpine-ta-elf32/lib

OR

* If your shell is 'csh' shell, set the environment variables for the installed toolchain. In case you have installed the ThreadArch toolchain to a different path, please specify that path instead of /opt/gnu-ta to the appropriate installed directory path.
* Type the following command on the command prompt.

Environment variables settings for Tool Chain:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

<prompt> setenv PATH /opt/gnu-ta/bin:/opt/gnu-ta/lib/gcc-lib/rpine-ta-elf32/3.3:$PATH

<prompt> setenv LD\_LIBRARY\_PATH /opt/gnu-ta/lib:/opt/gnu-ta/lib/gdb:/opt/gnu-ta/lib/gcc-lib:/opt/gnu-ta/rpine-ta-elf32/lib:$LD\_LIBRARY\_PATH

<prompt> rehash

**Note:**

'<prompt>' represents the command prompt

# About ThreadArch Toolchain Tools

The ThreadArch Toolchain consists of many tools such as Assembler, Compiler, Linker, Debugger, and Object dumper, etc. The Compiler, Assembler, Linker, and Debugger support the command line options supported by the base GNU C compiler, Assembler, Linker, and Debugger respectively. To display information about the toolchain, use the following commands.

<prompt>info -f <targetpath>/info/gcc.info

<prompt>info -f <targetpath>/info/as.info

<prompt>info -f <targetpath>/info/ld.info

<prompt>info -f <targetpath>/info/gdb.info

Here, <targetpath> - refers to the ThreadArch toolchain installed location.

The default installation directory for ThreadArch toolchain is /opt/gnu-ta

The following sections list the basic options for GCC, GDB, and Binutils

## GCC

ThreadArch GCC is used to compile (cross compile) programs written in ‘C’ language and generate a binary executable file to execute on the ThreadArch processor. GCC performs preprocessing, compiling, assembling, and linking the 'C' programs and generate a binary executable file.

The following table lists the basic command line options.

|  |  |
| --- | --- |
| **Option** | **Description** |
| -E | Stop after preprocessing the source file. Do not generate the assembly file. |
| -S | Stop after generating the assembly file. Do not assemble |
| -c | Stop after assembly. Do not link. |
| -g | Generate debugging information. |
| -o <file> | Generate the output in the file <file> |
| -Q | Makes the compiler print out each function name as it is compiled, and also print some statistics about each pass when it finishes. |

Table 1: GCC Command Line Options

Please refer to <http://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/> for more information about using the GNU Compiler.

## GDB

The Debugger allows you to see what is going on ‘inside’ another program while it executes or what another program was doing at the moment it crashed or terminated. The GDB helps:

* Stop the program on specified conditions
* Examine the state of the process when stopped
* Make changes in the process and enable debugging

ThreadArch GDB is used to execute and debug the binary executable file which is generated by ThreadArch GCC. It can execute and debug application on simulator as well as ThreadArch processor. In order to execute and debug applications on ThreadArch processor, the GDB uses the JTAG interface.

The following table lists the basic GDB commands:

|  |  |
| --- | --- |
| **Command** | **Description** |
| target sim | Use the simulator for debugging. |
| target pport | Use the remote target for debugging (remote target like FPGA, EVB etc). |
| load <file> | Load the file passed as argument to the debugger if this argument has not already been specified to gdb at startup. |
| run [arguments] | Start debugging the program loaded with the arguments specified. |
| Step | Execute the next program line (after stopping). Step into any function calls in the line. |
| break <opt> | Set the breakpoint at the function and line number in a source file specified as ‘opt’. |
| Print <exp> | Print the value of the expression specified as argument. As an example, if 'a' is a variable in the program being debugged and its value is 10, then ‘print a’ will display ‘a = 10’. If a register in ThreadArch has to be displayed, the command is ‘print $r0’ where r0 is the register to be displayed. |
| info reg <r> | Print the value in register <r> ex. info reg r0. It will print the value of register r0. |

Table 2: GDB Commands

|  |
| --- |
| **Note:**  In order to debug the program using GDB, compile with -g option. |

Please refer to <http://hubbard.engr.scu.edu/embedded/avr/doc/gnu/gdb/gdb.pdf> for more information about using the GNU Debugger.

On issuing ctrl+c command in debugger the processor will not halt and the execution of background processes continues. This feature was added as most of the applications are real time applications which require processor to be up all the time. If you want the processor to halt then you should introduce some software breakpoints and atleast one breakpoint should be registered in gdb so that when you press ctrl+c then the processor will halt.

## Binary Utilities (Binutils)

The GNU Binutils are a set of collection of binary tools. The main ones are:

|  |  |
| --- | --- |
| ld | The GNU linker. |
| as | The GNU assembler. |
| ar | A utility for creating, modifying, and extracting from archives. |
| gprof | Displays profiling information. |
| nm | Lists symbols from object files. |
| objcopy | Copies and translates object files. |
| objdump | Displays information from object files. |
| ranlib | Generates an index to the contents of an archive. |
| readelf | Displays information from any ELF format object file. |
| size | Lists the section sizes of an object or archive file. |
| strings | Lists printable strings from files. |
| Strip | Discards symbols. |

Table 3: Binutils Options

For more information on the usage and options for the above utilities please refer to <ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32/doc/binutils.pdf>

## Toolchain Flow

The following figure describes the steps involved in the generation of executable from C code and debugging the executable.



Figure 1: Toolchain Flow

# Compiling and Simulating

The following section explains the steps to be taken to compile and execute a C language application using the ThreadArch toolchain on its simulator.

Step-1: Set the environment for ThreadArch Toolchain.

Step-2: Write simple 'c' application.

Consider the following example, file.c:

int i = 1;

int j = 2;

main()

{

int k;

k = i+j;

return k;

}

Step-3: Compile the application using ThreadArch compiler.

For example:

<prompt>rpine-ta-elf32-gcc -g file.c -o test

Where ‘rpine-ta-elf32-gcc’ is ThreadArch compiler,

‘file.c’ is source file which is require to be compiled,

‘test’ Binary executable file.

|  |
| --- |
| Note: In order to debug applications using GDB, it is required to compile the application using -g option which generates debugging information. |

Step-4: Execute the application (Binary executable) by invoking GDB.

For example

<prompt>rpine-ta-elf32-gdb test

Where ‘rpine-ta-elf32-gdb’ is ThreadArch debugger,

‘test’ is the binary executable file.

In order to execute or debug application using ThreadArch debugger on simulator, give the following commands:

(gdb) target sim

---Connect the simulator

(gdb) load

--- Load the binary executable file into simulator.

(gdb) run

--- Starts running the loaded file

You can execute and debug applications by using gdb commands such as run, break, continue, step, next, print, and info, etc.

For more information about gdb command refer to <http://hubbard.engr.scu.edu/embedded/avr/doc/gnu/gdb/gdb.pdf>

(gdb) quit

---Quit from debugger after completion of execution or debugging

|  |
| --- |
| Note: Please refer to application1 in the ‘Applications’ directory in the delivered database. |

# Compiling and Running a Program on a Target

This section describes the procedure involved in compiling a program and running it by downloading it into a target board (FPGA, chip, etc.) via JTAG. The debugger is also active in this setup. The target board should contain the ThreadArch processor. The board must have a JTAG port that is connected to the JTAG port on the target. A parallel cable is then to be used to connect the board, via its JTAG connector, to the host system’s parallel port.

Step-1: Set the environment for ThreadArch.

Step-2: Create a sample test case ‘file.c’ as below.

int i=1;

int j=2;

main()

{

int k;

k=i+j;

return k;

}

Step-3: Compile the test program using ThreadArch toolchain.

<prompt> rpine-ta-elf32-gcc -g file.c ivt.s -T rpineelf.x -o test

Where

|  |  |
| --- | --- |
| ivt.s | The assembly language routine containing the interrupt vector table. |
| rpineelf.x | A linker script. |

|  |
| --- |
| Note: Modify the above files according to the application requirements. |

Step-4: Execute the application as below by invoking GDB.

<prompt> rpine-ta-elf32-gdb test

Step-5: issue following command in gdb

(gdb) target pport

---This makes the ThreadArch processor as the remote target. Connect the board to the host system using JTAG through parallel port ‘lp0’.

(gdb) load

--- load the binary executable onto target

(gdb) set \*0x22000004=0

---Reset the processor (writing 0 to this address translates releases the processor from a soft reset).

(gdb) run

---starts execution on target

(gdb) quit

|  |
| --- |
| **Note:** Program control will be in a loop on completion of execution. To come out from gdb, press Ctrl+C and issue quit command. |

|  |
| --- |
| **Note:**  Please refer to application2 in the ‘Applications’ directory in the delivered database. |

# Compiling and Running a Multi-Threaded Program

This section explains the steps needed to compile and execute a multi-threaded application using ThreadArch Toolchain. In the program mentioned below, a dual-threaded application is given.

Programmer needs to program the scheduler\_thread\_id\_vector register. It is 32-bit wide having 16-slots to carry thread ids. Each slot corresponds to 2-bits to represent one of the 4 threads. Decide the percentage of MIPS to be given to each thread (please refer ThreadArch\_hardware\_reference\_manual for more information).

For example, in those 16 slots, if thread 1 was programmed in 4 slots, thread 2 was programmed in 8 slots and thread 3 in 4 slots. It means that thread 0 has 0% of MIPS, thread 1 & 3 have 25% of MIPS each, and thread 2 has the remaining 50% of MIPS.

The example mentioned below is programmed to have 75% of MIPS in thread 0 and 25% of MIPS in thread 1.

Step-1: Set the environment for ThreadArch.

Step-2: Create a sample test case ‘file.c’ as below.

#define SCHEDULER\_TH\_IDS\_VEC\_REG 0x22000060

#define CONFIGURED\_VALUE\_IN\_REG 0x01010101



Figure 2: Multi threaded program

Note:

00 in slot indicates Thread 0

01 in slot indicates Thread 1

10 in slot indicates Thread 2

11 in slot indicates Thread 3

char s3[15];

int main()

{

\*(int \*)SCHEDULER\_TH\_IDS\_VEC\_REG = (int) CONFIGURED\_VALUE\_IN\_REG;

char \*s1="RDIEINL";

int i,k=0;

for(i=0;s1[i];i++)

{

s3[k]=s1[i];

k=k+2;

}

return 0;

}

int main\_th1()

{

char \*s2="EPNSGAS";

int i,k=1;

for(i=0;s2[i];i++)

{

s3[k]=s2[i];

k=k+2;

}

return 0;

}

Step-3: Compile the test case using GNU tools for ThreadArch.

<prompt> rpine-ta-elf32-gcc -g file.c ivt.s start\_th1.s -T rpineelf.x -o test

Where

|  |  |
| --- | --- |
| ivt.s | The assembly language routine containing the interrupt vector table. |
| rpineelf.x | A linker script. |
| start\_th1.s | The assembly language rountine containing Thread1 startup code. |

|  |
| --- |
| Note: Modify these files according to the application requirements. |

Step-4: Execute the application as below by invoking GDB.

<prompt> rpine-ta-elf32-gdb test

Step-5: Execute the application by invoking GDB as described in the previous sections.

|  |
| --- |
| Note: Please refer to application3 in the ‘Applications’ directory in the delivered database. |

# Create a library

This section explains the steps to create a library and use it in a different function.

Step-1: Set the environment for ThreadArch toolchain.

Step-2: Create the first file ‘file1.c’ as below.

int add(int i,int j)

{

int k;

k= i + j;

return k;

}

Step-3: Create the second file ‘file2.c’ as below.

int subtract(int a,int b)

{

int c;

c= a - b;

return c;

}

Step -4: Create the object files using ThreadArch™ compiler

<prompt> rpine-ta-elf32-gcc -c file1.c –o add.o

<prompt> rpine-ta-elf32-gcc -c file2.c -o subtract.o

Step-5: Archive the above object files into a single library file as below

<prompt> rpine-ta-elf32-ar -qv libmy.a subtract.o add.o

Step-6: Create a ‘main’ file which uses the above functions in the library

main()

{

int a,b,c,d;

a=b=10;

c = add(a,b);

d = subtract(a,b);

return 0;

}

Step-7: Link the library to the main file and generate an object file as below.

<prompt> rpine-ta-elf32-gcc -g main.c libmy.a -o test

|  |
| --- |
| **Note:** Please refer application4 in the ‘Applications’ directory from the delivered database. |

# Introduction to Optimization

Optimization options affect the behavior of GCC. The following are the various options that control Optimization.

'-O' or '-O1'

Optimize. Optimizing compilation takes lot of time and a lot of memory for a large function. With -O, the compiler tries to reduce code size and execution time without performing any optimizations that take a great deal of compilation time.

`-O2'

Optimize even more. GCC performs nearly all the supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify -O2. As compared to -O, this option increases compilation time as well as the performance of the generated code.

`-O3'

Optimize yet more. `-O3' turns on all the optimizations specified by `-O2' and also turns on the `-finline-functions' and `-frename-registers' options.

`-O0'

Do not optimize.

`-Os'

Optimize for size. `-Os' enables all `-O2' optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

If you use multiple `-O' options, with or without level numbers, the last such option is the one that is effective.

# Basic Application Binary Interface Specification

## Scope

This document specifies the Application Binary Interface (ABI) for the ThreadArch Processor and is provided as a reference for application developers. Compiler and Assembler developers also may use this ABI standard while creating tools for the ThreadArch architecture. This document covers run-time aspects as well as object formats to be used by tool chain.

This document covers only ThreadArch processor specific information for GNU ThreadArch Compiler toolkit. For more information about GNU tools such as compiler (gcc), assembler (as), linker (ld), and debugger (gdb), please refer to the informative files present in the Threadarch toolchain installation directory. This can be done as follows.

<prompt>info -f <targetpath>/info/gcc.info

<prompt>info -f <targetpath>/info/as.info

<prompt>info -f <targetpath>/info/ld.info

<prompt>info -f <targetpath>/info/gdb.info

Where,

<targetpath> - refers to ThreadArch toolchain installed location

The default installation directory for ThreadArch toolchain is /opt/gnu-ta

## Compiler

### Command Line Options

GCC for ThreadArch supports the command line options supported by the base GNU C Compiler. Please refer to the standard ‘gcc’ information.

### Data Types and Arguments

The following table shows the data types, their sizes, and memory alignments for the ThreadArch Processor.

|  |  |  |
| --- | --- | --- |
| **Type** | **Size (Bytes)** | **Alignment (Bytes)** |
| Signed char | 1 | 4 |
| Unsigned char | 1 | 4 |
| Short | 2 | 4 |
| Int | 4 | 4 |
| Unsigned Int | 4 | 4 |
| Long | 4 | 4 |
| Long Long | 8 | 4 |
| Unsigned long long | 8 | 4 |

Table 4: Data types

|  |
| --- |
| Note: All the data types are stored in little-endian format (that is, lower bytes first, then higher bytes). |

### Registers

ThreadArch processor contains 16 data/address registers and 4 control registers. All the registers are 32-bit wide.

All the data/address registers can be used as general purpose registers. The register ‘R15’ is used as the stack pointer and the register ‘R14’ is used as the frame pointer.

The register names and their description are shown in the following table.

|  |  |
| --- | --- |
| **Register Name** | **Register Description** |
| r0-r15 | Data/Address Register. |
| sr | Status Register. |
| pc | Program Counter Register. |
| M01 | Circular mode control register |
| ier | Interrupt Enable Register. |
| r0-r15 | Data/Address Register. |
| sr | Status Register. |

Table 5: Regsiters

## Calling Conventions

The ThreadArch processor uses a grow-down stack towards the lower address and contains return address, local variables, and the parameters passed to a routine etc. The registers ’r0’,’r1’,’r2’,’r3’, ’r4’,and ’r5’ are used as caller-saved registers and all the other registers are used as callee-saved registers (except stack pointer ‘r15’).

### Stack Allocation

The stack space required by a called function is allocated by the function’s prologue code. Stack allocation before the function call and after the function call are shown in the following figures:



Figure 3: Stack allocation before a fucntion call



Figure 4: Stack allocation after a function call

For example:

Int main( )

{

int a,b,c,d,e,f;

a = 10;

b = 20;

c = 30;

d = 40;

e = 50;

f = 60;

foo(a,b,c,d,e,f);

}

int foo(int k,int l,int m, int n, int o,int p)

{

….

}

### Stack Before Calling Function foo()



### Stack after Calling the Function foo()



Note: [] -indicate value of the register or variable

Note: Function arguments are passed through registers. First 4 arguments can be passed through registers (r0,r1,r2,r3)(caller saved) and the rest through stack.

ThreadArch processor stack uses the stack pointer (r15) register to access the stack.

Function Prologue and Epilogue:

The called routine is responsible for allocating its own stack frame. This action is performed by the ’Prologue’ code which the compiler places before the body of the routine. After the body of the routine, the compiler generates an Epilogue to restore the processor to the state it was prior to the prologue.

The Prologue code does the following:

* Store the content of the registers that are to be used in the routine.
* Allocates space for the function arguments.
* Increments the stack pointer to account for the new stack frame.

At the end of the function, the Epilogue does the following:

* Restore the registers contents that stored in prologue.
* Restores the stack pointer to the beginning of function as it is before prologue.
* Restores the return address register.

### Passing Arguments

Arguments to functions are passed through registers. Four (4) arguments can be passed through registers (caller saved) and the rest through stack.

### Function Return

All the function values are returned using register ‘r0’ and ‘r1’. r1 is used when return value is long long.

## Assembler

### ThreadArch Assembler Syntax

The following table describes the special characters used in the ThreadArch assembler.

|  |  |
| --- | --- |
| **Char** | **Description** |
| ‘#’ | In an operand, it represents an immediate value. |
| ‘;’ | In a line, it represents a comment and rest of the line is ignored. |
| ‘/\* \*/’ | Multi-line comments |

Table 6: Assembler Syntax

### Directives

There are no ThreadArch specific directives required for ThreadArch Assembler.

## Object File Formats

ThreadArch tools use ELF 2.0 object file formats and DWARF 2 debugging information formats.

### Sections Information

The Object dumper (rpine-ta-elf32-objdump) with the following options provides the required section information.

-h, --[section-]headers

Displays the contents of the section headers

-d, --disassemble

Displays assembler contents of executable sections

-s, --full-contents

Displays the full contents of all sections requested

-S, --source

Displays source code intermixed with disassembly, if possible. Implies -d.

-j, --section=NAME

Only displays contents of section NAME

Please refer to the following link for the objdump options:

<ftp://ftp.estec.esa.nl/pub/ws/wsd/erc32/doc/binutils.pdf>

### Symbol Table Format

There are no ThreadArch processor-specific symbol table requirements.

## Debugger

### Connecting to Simulator

To connect to the ThreadArch simulator, type the following command after invoking the ThreadArch Debugger.

(gdb) target sim

The above command will connect to the ThreadArch simulator.

### Connecting to Target

To connect to the target, type the following command after invoking the ThreadArch Debugger.

You need to run script insert.sh present in gnu-ta/bin/ directory before invoking ThreadArch Debugger.

(gdb) target usbport

The above command will connect to the target through Jtag.

Run script remove.sh present in gnu-ta/bin/ directory after returning from ThreadArch Debugger.

Script insert.sh installs the driver required to support usb debugger module. If you are compiling the driver from the source you can insert the kernel module from build path as well.

### Command Line Options

There are no ThreadArch specific options required for the debugger.

### Debugging Command

ppe\_dumpio – dumps all the registers present on PPE IO bus.

ppe\_dumprx – dumps all LMAC RX path registers.

Ppe\_dumptx – dumps all LMAC TX path registers.

ppe\_readio <index in decimal> - reads a particular register.

Ppe\_readrx <index in decimal>

Ppe\_readtx <index in decimal>

ppe\_break – puts PPE in hold.

Ppe\_resume – resumes PPE.

Ppe\_ibp <pc address in hex without 0X prefixed> - sets break point at given PC.

Ppe\_dbp <encoded address in hex with 0X prefixed> - upper most nibble in 32 bit address defines the access rules. Encoding is as follows.

Upper nibble 0 – access break point.

1 – Write.

2 – Read.

Ppe\_delete <address in hex> - deletes break point (data or instruction).

Ppe\_break\_info – displays all set break point.

Ppe\_pc – current ppe\_pc and stack pointer

Ppe\_inforeg – ppe\_internal registers.

### Compiling the driver Source

Go to <ToolChain-Path>/usb\_driver.

Do Make clean; make

Use script start.sh to insert driver module.

**Revision History**

| **Version No.** | **Date** | **Changes** |
| --- | --- | --- |
| 1.0 | June 2018 | Initial Version |