RSDK Toolchain User Guide

Realtek Semiconductor Corp.
Release 4.6.4
October 31, 2014

RSDK Toolchain User Guide for Release 4.6.4

Contents

List of Tables

Chapter 1 RSDK

Introduction

RSDK, Realtek Software Development Kit, is a software toolchain that empowers end-users to develop embedded applications that run on Realtek's in-house processor cores. The development kit includes binary utilities (such as assembler and linker), compiler, C libraries, and debugger. The binary utilities, compiler, and debugger are derived from GNU Binutils, GCC (the GNU Compiler Collection), and GDB (the GNU Project Debugger) respectively. RSDK supports two kinds of C libraries - newlib and uClibc, and they are delivered with different RSDK tarballs. Newlib-based RSDK is used for the development of deeply embedded application and RTOS (Real-Time Operation System), with target prefix 'rsdk-elf'. uClibc-based RSDK is used for Linux development environment, with target prefix 'rsdk-linux'.

Version Numbering

The format of RSDK version number is shown as follows:

```
GNU version (Realtek <RSDK version>p<patch level> Build <build number>)
```

The meaning of the version number are listed as below:

- GNU version - the version number of GNU toolkit which RSDK is based on, such as binutils-2.22.0. This number is only updated with the GNU source of RSDK, and the update of GNU source usually leads to better performance.
- RSDK version - the main version number of RSDK toolchain, which contains three numbers. The first and second version number is the major version number of toolchain generation, which are updated only when we upgrade the code base of GNU sources, such as upgrading GCC-4.4 to GCC-4.6. The third version number is updated when there are large new features are added.
- patch level - this number is updated for major bug fix or improvement.
- build number - this number is incrementally updated automactically with the toolchain development, so the number represents a development state of the toolchain sources.

An example of RSDK version number is as Program 1.

Software Component

The list of software components and their version numbers is summarized in table 1.1.

Example 1: RSDK Version number

```
% rsdk-elf-gcc -v
Using built-in specs.
Target: mips-elf
Configured with: RSDK Builder release 4.6
Thread model: single
gcc version 4.6.3 (Realtek RSDK-4.6.3 Build 259)
```

Table 1.1: Software Components

Software	Original Version
gcc	4.6.3
binutils	2.22.0
gdb	7.2
newlib	1.20.0
uClibc	0.9.33

Supported Processor Cores

The list of supported processor cores is summarized in table 1.2.

Table 1.2: Supported LX/RLX CPU cores

Processor Core	RTL Release Version	MIPS1	MIPS16	RADIAX
RX3081	1.0	No	Yes	No
RLX4081	1.0	Yes	Yes	No
LX4180	4.0	Yes	Yes	No
LX5280	1.4	Yes	Yes	Yes
RLX4181	1.5	Yes	Yes	No
RLX5181	1.6	Yes	Yes	Yes
RX4281	1.4	Yes	Yes	No
RX5281	1.4	Yes	Yes	Yes

Supported Environment

The list of supported environment is summarized in table 1.3.

Table 1.3: Supported RSDK platforms

Platform	Version	Package name
Linux	RedHat Enterprise Linux 4 and above	rsdk-4.6.4 -linux.tar.gz
Cygwin	Cygwin 1.5.10 and above	rsdk-4.6.4 -cygwin.tar.gz

The C library shipped with RedHat Linux may differ from the one the RSDK toolchain was built against. To ensure maximal compability, the following two packages are recommended if the RedHat Linux version is newer than 7.3.

compat-glibc-7.x-2.2.4.32.6
compat-libstdc++-7.3-2.96.128

NOTE: The Cygwin platform itself is not a stable environment for software development. Users might encounter various problems which are not directly related to the RSDK toolchains. To ensure maximal stability, users are advised to develop applications on Linux platforms whenever possible.

Supported C Libraries

The list of supported C libraries is summarized in table 1.4.

Table 1.4: Supported C Libraries

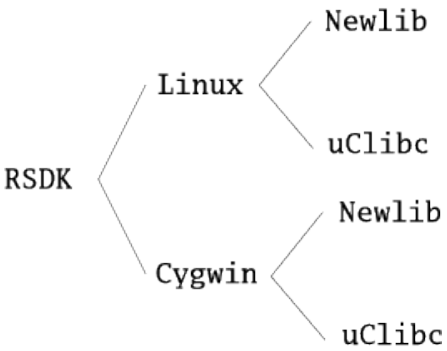
Library	Version
Newlib	1.20.0
uClibc	0.9.33

Installation

The RSDK packages are available as four tarballs, one for each platform and each C library. These toolchains are completely independent. Users should only need to download the desired RSDK toolchain and point the executable path to where it is installed. The tarballs are as follows:

rsdk-4.6.4 -newlib-linux.tar.gz
rsdk-4.6.4 -uclibc-linux.tar.gz
rsdk-4.6.4 -newlib-cygwin.tar.gz

The structure of the RSDK tarball is shown as follows:



The installation procedure is shown in program 2. The list of supported libraries and their versions is summarized in table 1.4.

Example 2: RSDK installation procedure

```
step 1: cd TARGET_DIR
step 2: tar jxvf rsdk-{VERSION}-{LIBRARY}-{PLATFORM}.tar.bz2
step 3: ln -s rsdk-{VERSION}/{PLATFORM}/{LIBRARY} rsdk
step 4: set path=(TARGET_DIR/rsdk/bin $path)
```

Chapter 2 GCC

GCC (the GNU Compiler Collection) includes front ends for multiple languages such as C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages (libstdc++, libgcj,...).

In RSDK 4.6.4 , the GCC compiler is derived from GCC-4.6.3 with C and C++ language supports. There are also changes for the better supports of software development, and these changes are listed in the following subsections.

General Changes

Relative path search

GCC search certain paths for binaries, libraries, and includes files during compilation. In RSDK, to ensure maximal portability, GCC has been patched to search paths relative to the GCC binary.

Default options

The following options, listed in table 2.1, are enabled by default.

Table 2.1: Default compiler options

Options	Description
-msoft-float	Enable software floating point support
-EB	Enable big-endian
-march=4180	Set default target to LX4180 if none is specified

Machine-dependent options

-march=3081|4081|4180|4181|4281|5181|5280|5281

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Specify target processor core
Status: Current
Version: 1.5.9
Description:

The march|mtune|mcpu option sets the target processor core to the one specified in the option.
If none of the -march, -mtune, and -mcpu options is specified, compiler will automatically add -march=4180 to the option list.

-mt0-t3

Architecture: RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Expand function parameter registers from four to eight
Dependency: None
Status: Current
Version: 1.2.0+
Description:

By MIPS ABI convention, only four registers, a0, a1, a2, and a3, are used to pass function parameters. When calling a function with more than four parameters, extra parameters are pushed into and popped out of the stack before and after entering the callee. There are two major drawbacks for this approach. First, incremented number of memory accesses for loading and storing these parameters will certainly degrade the performance. Second, loading data from memory has the load delay penalty and may incur cache miss, which makes things even worse. Therefore, it is beneficial to expand the number of function parameter passing registers from four to eight at the cost of breaching ABI compatibility. When -mt0-t3 option is specified, compiler will use four additional registers, t0, t1, t2, and t3, for passing function parameters. The total number of registers that are reserved for passing parameters is increased from four to eight.

NOTE: this option is not ABI compatible. If this option is used, it should be applied to all the source files in the application.

NOTE: When using inline assembly with this option enabled, users must take caution not to destroy the extra registers, t0-t4. These extra registers should be saved first if they will be used in the inline assembly codes and should be restored after use.

-msimd

Architecture: RLX5181, LX5280, RX5281
Summary: Expand single instruction multiple data support
Dependency: -mradiax
Status: Current
Version: 1.2.0+
Description:

RLX5181, LX5280, and RX5281 support Single Instruction Multiple Data (SIMD) instructions. A SIMD instruction operates on multiple values contained in a single register at the same time.

Table 2.2: Built-in data type for SIMD instructions

Type	Definition	Internal Type
v2hi	typedef int v2hi __attribute__((mode(V2HI))	VNB

For multa2, mulna2, madda2, and msuba2, there are three different forms for their builtin functions. The three different forms are internal1, internal2, and internal3. For internal1, the destination register is the upper 32-bit HI of the accumulator. For internal2, the destination register is the lower 32-bit LO of the accumulator. For internal3, the destination register is the entire 64-bit accumulator, HI and LO.

-msave-restore-mmd

Architecture: RLX5181, LX5280, RX5281
Summary: Preserve MMD state
Dependency: -mradiax
Status: Current
Version: 1.2.0+
Description:

Table 2.3: Built-in functions defined for SIMD instructions

Function	Argument 0	Argument 1	Return Type
__builtin_absr2	V2HI	V2HI	V2HI
__builtin_addr2	V2HI	V2HI	V2HI
__builtin_subr2	V2HI	V2HI	V2HI
__builtin_min2	V2HI	V2HI	V2HI
__builtin_max2	V2HI	V2HI	V2HI
__builtin_multa2_internal1	V2HI	V2HI	V2HI
__builtin_multa2_internal2	V2HI	V2HI	V2HI
__builtin_multa2_internal3	V2HI	V2HI	DI (long long)
__builtin_mulna2_internal1	V2HI	V2HI	V2HI
__builtin_mulna2_internal2	V2HI	V2HI	V2HI
__builtin_mulna2_internal3	V2HI	V2HI	DI (long long)
__builtin_madda2_internal1	V2HI	V2HI	V2HI
__builtin_madda2_internal2	V2HI	V2HI	V2HI
__builtin_madda2_internal3	V2HI	V2HI	DI (long long)
__builtin_msuba2_internal1	V2HI	V2HI	V2HI
__builtin_msuba2_internal2	V2HI	V2HI	V2HI
__builtin_msuba2_internal3	V2HI	V2HI	DI (long long)
__builtin_sltr2	V2HI	SI (int)	V2HI
__builtin_sllv2	V2HI	SI (int)	V2HI
__builtin_srlv2	V2HI	SI (int)	V2HI
__builtin_srav2	V2HI	SI (int)	V2HI

MMD is a special register that is shared among many RADIAX instructions. The state of this MMD register is crucial for two reasons: First, compiler depends on the state of this MMD register to emit the right RADIAX instruction. Second, sequence of RADIAX instructions rely on the state of this MMD register to operate correctly. During compilation, the compiler keeps track of the state of MMD register carefully. However, if users change the state of the MMD register manually, for example, loading data into the MMD register in inline assembly codes, the result might be unpredictable. The -msave-restore-mmd is provided to add a safe net under this situation. When -msave-restore-mmd is specified, compiler will automatically emit instructions that save the state of MMD register before the operation that changes its state and emit instructions that restore the state of MMD register after the operation result is retrieved.

-mradiax

Architecture: RLX5181, LX5280, RX5281
Summary: Enable RADIAX support for RLX5181, LX5280, and RX5281
Dependency: None
Status: Current
Version: 1.2.0+
Description:

The -mradiax option enables the RADIAX support for RLX5181 and LX5280. The RADIAX instruction extensions include MAC operations, vector-addressing, and enhanced extensions to the MIPS-I ALU instructions.
For RLX5181 and LX5280, -mradiax automatically implies -mmac by default.

Example 3: Example of using built-in functions

```
int func2(int i)
{
    return i > 0 ? i : -i;
}

typedef short v2hi __attribute__ ((vector_size (4)));

int func1()
{
    int shift_num;

    v2hi data[32];
    v2hi sumv1, sumv2;
    long long sumv3;
    int sumv0 = 0;
    int i;
    v2hi a = {1,2};
    v2hi b = {2,1};

    for (i = 0; i < 32; i++)
        data[i] = __builtin_addr2((v2hi) i, (v2hi) i);

    for (i = 0; i < 32; i++) {
        sumv1 = __builtin_madda2_internal1(data[i], data[i]);
        sumv2 = __builtin_madda2_internal2(data[i], data[i]);
        sumv3 = __builtin_madda2_internal3(sumv2, data[i]);
    }

    shift_num = func2(-6);
    sumv2 = __builtin_srav2(sumv2, shift_num);
    return (int) __builtin_subr2(sumv2, (v2hi) sumv0);
}
```

-mmac

Architecture: RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Enable optional Multiply/Divide/Accumulator support
Dependency: None
Status: Current
Version: 1.2.0+
Description:

All the LX/RLX processor cores support an optional Multiply/Divide/Accumulate module (MAC-DIV) which further enhances mathematical operations. This MAC-DIV module is configurable using the lconfig utility. When -mmac option is specified, compiler will emit the instructions listed in table 2.4 whenever possible. It is users' responsibilities to ensure the MAC-DIV module exists in the target processor core.
The list of instructions for the optional MAC-DIV module is summarized as follows:

For RLX5181 and LX5280, -mradiax automatically implies -mmac by default.

-mcode-readable=yes/pcrel/no

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Specify when instructions are allowed to access code
Dependency: None
Status: Current
Version: 1.5.0+

Table 2.4: Optional MAC-DIV instructions

Mnemonic	Operation	Latency	Repeat Delay	Description
MTHI	HI <- Rs	-	-	pre-load accumulator, or restore saved HI
MTLO	LO <- Rs	-	-	pre-load accumulator, or restore saved LO
MFHI	Rd <- HI	1	-	read accumulator, or part of 64-bit result
MFLO	Rd <- LO	1	-	read accumulator, or part of 64-bit result
MULT	HI,LO <- Rs*Rt	5	-	32x32 signed multiply 64-bit result
MULTU	HI,LO <- Rs*Rt	5	-	32x32 unsigned multiply 64-bit result
MADH	HI <- HI+Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, with 32-bit signed add to accum
MADL	LO <- LO+Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, with 32-bit signed add to accum
MAZH	HI <- 0+Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, add to pre-zeroed 32-bit accum
MAZL	LO <- 0+Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, add to pre-zeroed 32-bit accum
MSBH	HI <- HI-Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, with 32-bit signed sub from accum
MSBL	LO <- LO-Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, with 32-bit signed sub from accum
MSZH	HI <- 0-Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, sub from pre-zeroed 32-bit accum
MSZL	LO <- 0-Rs[15:0]*Rt[15:0]	3	0	16x16 signed multiply, sub from pre-zeroed 32-bit accum
DIV	HI <- Rs%Rt; LO<-Rs/Rt	35	-	32 by 32 signed divide with reminder
DIVU	HI <- Rs%Rt; LO<-Rs/Rt	35	-	32 by 32 unsigned divide with reminder

Description:

Specify whether GCC may generate code that reads from executable sections. There are three possible settings:

- **-mcode-readable=yes** Instructions may freely access executable sections. This is the default setting.
- **-mcode-readable=pcrel** MIPS16 PC-relative load instructions can access executable sections, but other instructions must not do so.
- **-mcode-readable=no** Instructions must not access executable sections. This option can be useful on targets that are configured to have a dual instruction/data SRAM interface but that do not automatically redirect PC-relative loads to the instruction RAM.

NOTE: RX3081 set “-mcode-readable=no” by default.

-mgpopt|-mno-gpopt

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Use GP-relative addressing to access small data
Dependency: ‘-G’, ‘-mlocal-sdata’, ‘-mextern-sdata’ **Status:** Current
Version: 1.5.0+
Description:

Use (do not use) GP-relative accesses for symbols that are known to be in a small data section; the related options are ‘-G’, ‘-mlocal-sdata’ and ‘-mextern-sdata’. ‘-mgpopt’ is the default for all configurations beside RX3081 (due to lack of `$gp`).

‘-mno-gpopt’ is useful for cases where the `$gp` register might not hold the value of symbol `_gp`. For example, if the code is part of a library that might be used in a boot monitor, programs that call boot monitor routines will pass an unknown value in `$gp`. (In such situations, the boot monitor itself would usually be compiled with ‘-G0’.)

‘-mno-gpopt’ implies ‘-mno-local-sdata’ and ‘-mno-extern-sdata’. The -mgpopt switch says to write all of the data declarations before the instructions in the text section, this allows the MIPS assembler to generate one word memory references instead of using two words for short global or static data items. This is on by default if optimization is selected.

Compiler Options

-ftword

Architecture: RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Enable load/store twin-word instruction, ltw or lt/st
Dependency: None
Status: Current
Version: 1.2.4+
Description:

When -ftword is specified, compiler will emit ltw (RLX4181) instruction or lt (RLX5181/LX5280) instruction instead of a sequence of load byte and shift instructions whenever possible. Likewise, when -ftword is specified, compiler will emit st (RLX5181/LX5280) instruction instead of a sequence of store byte and shift instructions whenever possible. These options are added since RSDK 1.2.4.

-ftword-stack

Architecture: RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Enable twin-word instructions in function prologue/epilogue
Dependency: None
Status: Current
Version: 1.2.4+
Description:

When -ftword-stack is specified, compiler will emit lt/ltw instruction instead of a sequence of load byte and shift instructions whenever possible during function calling. This option is added since RSDK 1.2.0.

-frlxcov

Architecture: RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Put the code coverage initialization symbols in section `.rlxcov` instead of `.ctors`
Dependency: None
Status: Current
Version: 1.5.0+
Description:

When -frlxcov is specified, compiler will emit codes to do coverage analysis for basic blocks. This option is similar to the combination of '-fprofile-arcs -ftest-coverage' except that it uses the RSDK supplemental library which supports remote file I/O over GDB remote serial protocol.

The coverage analysis codes will be placed in **.rlxcov** section. Therefore, the linker script must be modified to explicitly allocate the **.rlxcov** section.

The '-fprofile-arcs -ftest-coverage' options have been reverted to comply with GNU standard. If specified, the standard GNU code coverage analysis will be used. Please refer to GNU website for more information.

-fdafile-relative

Architecture: RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Put gcda file in relative instead of absolute path
Dependency: -fprofile-arcs -ftest-coverage
Status: New
Version: 1.2.7+
Description:

By default, GCOV generates .da file in the path where the source files reside. When this option is specified, GCOV will generate .da file in the path where the executable is invoked.

subsection*-fmerge-constants **Architecture:** LX4180, RLX4181, RLX5181, LX5280
Summary: Attempt to merge identical string constants across compilation units
Dependency: -O and above
Status: Current
Version: 1.5.0+
Description:

If this option is enabled, GCC will attempt to merge identical string literals and float point constants across compilation units by putting string literals and/or floating point constants in dedicate sections **.rodata.str1.4** and **.rodata.cst4**. The benefit is that the code size can be reduced because duplicate constants are removed. For Linux kernel 2.4, the magnitude of reduction in code size and in memory footprint is in the order of mega bytes.

In GCC version 4, this option is turned on by default for optimized compilation if the assembler and linker support it. This option was enabled at levels -O, -O2, -O3, and -Os. In other words, if optimization is turned on, the compiler will remove duplicate constants by merging them into dedicate sections.

NOTE: If this option is turned on, the linker script must explicitly include the two dedicate sections .rodata.str1.4 and .rodata.cst4 if they are not already dealt with. Example is shown as follows:

```
. . . .

.data    :
{
  _fdata = . ;
  *(.data)

  *(.rodata.cst4)             /* merged constant */
  *(.rodata.str1.4)           /* merged string literals */

 /* Align the initial ramdisk image (INITRD) on page boundaries. */
 . = ALIGN(4096);
 __rd_start = .;
 *(.initrd)
 __rd_end = .;
 . = ALIGN(4096);

  CONSTRUCTORS
}

. . . .
```

-fuse-uls

Architecture: RLX4181, RX4281, RLX5181, RX5281
Summary: Enable unaligned load/store instructions
Dependency: None
Status: Current
Version: 1.5.0+
Description:

When this option is specified, GCC will generate unaligned load/store instructions whenever possible.

-ffix-bdsl

Architecture: RX4281, RX5281
Summary: Fix Branch-Delay-Slot-Load (BDSL) issue

Dependency: None
Status: Current
Version: 1.5.5p1+
Description:

This option is only for Taroko processors (prior to v1.3) to fix the known issues: Branch-Delay-Slot-Load (BDSL) and immediate boundary problem. When this option is specified, the compiler will do following fix and detection:

- Avoid to load instruction in branch delay slot at mips1 mode.
- Avoid to put any instruction in branch delay slot at mips16 mode.
- Raises warning to user if BDSL issue occur at noreorder section (assembly).
- Auto-extend mips16 instruction as 32-bit for inappropriate immediate boundary value.

-fuse-balx

Architecture: RX4281, RX5281
Summary: Use bx/balx for fast ISA mode switch
Dependency: None
Status: New
Version: 1.5.9+
Description:

This option enable the use of bx/balx instructions for fast ISA mode switch, which is supported only for Taroko processors posterior to v1.4.

Warning Options

-Wpossible-load-use

Architecture: RLX4081, LX4180, RLX4181, RLX5181
Summary: Warning for potential load-use
Dependency: None
Status: New
Version: 1.5.5p3+
Description:

When this option is specified, the assembler outputs warning for the load instruction at branch delay slot. For the processors without hardware-interlock support, a load instruction at branch delay slot, it may be used by the next instruction at the target address. Therefore these load instructions have potential load-use issue.

-Wmissing-delay-slot

Architecture: RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Warning for missing delay slot
Dependency: None
Status: New
Version: 1.5.7
Description:

When this option is specified, the compiler passes '-warn-missing-delay-slot' option to the assembler, and the assembler outputs warning for the load instruction at the end of assembly fragment (such as the end of section, before a new directive). This means, check the missing instruction for delay slot for user.

Profiling Options

-pg

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Enable general program profiling
Dependency: None
Status: Current
Version: 1.5.0+
Description:

When this option is specified, compiler will emit instructions in functions prologue and epilogue to jump to the predefined profiling functions for general program profiling, e.g. user applications. These two options are identical.

Attributes

__attribute__((far_call))

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Mark the specified function as a long jump
Dependency: None
Status: Current
Version: 1.2.0+
Description:

By default, the range for a jump instruction is within 256MB. If the jump target is more than 256MB away, then a single jump instruction can not reach the desired target. In this case, the jump instruction must be modified to a load and a jump instructions. The former loads the target address to a specific register and the later does the actual jump to the address stored in that register. By using the far_call attribute, users can explicitly specify certain functions as long jumps and compiler will emit the right instructions when these functions are called.

Program 1: Example of using __attribute__((far_call))

```
int func() __attribute__((far_call))

int func()
{
    ....
}

int myfunc()
{
    ....
    func();
}
```

The purpose of attribute is similar to that of the compiler option -mlong-calls. The difference is that the compiler option, -mlong-calls, applies to all the functions in the application, while __attribute__((far_call)) allows users to do finer control and apply to specific functions only.

__attribute__((mips16))
__attribute__((nomips16))

Architecture: RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Mark the specified function to be compiled in MIPS16/NONMIPS16 mode

Dependency: None
Status: Current
Version: 1.5.0+
Description:

The __attribute__((mips16)) enables users to insert MIPS16 codes into MIPS1 applications without compiling the entire source as a MIPS16 code. In other words, with this attribute, users can fine control certain functions to be in MIPS16 mode and balance the trade-off between code performance and code size.

Likewise, the __attribute__((nomips16)) enables users to insert MIPS1 codes into MIPS16 applications without compiling the entire source as a MIPS1 code.

The example is shown in program 2.

Program 2: Example of using __attribute__((mips16))

```
int __attribute__((mips16)) func1()
{
    ....
}

int func2()
{
    ....
}

int myfunc()
{
    ....     /* MIPS32 mode */
    func1(); /* MIPS16 mode */
    func2(); /* MIPS32 mode */
    ....     /* MIPS32 mode */
}
```

NOTE: RX3081 is always on MIPS16 mode.

NOTE: the compilation time will increase as the number of functions with MIPS16 attribute increases. The overhead is introduced by testing and switching between MIPS1 and MIPS16 mode on a per function basis. If the majority of the functions in a C file are MIPS16, users should consider compiling the entire file in MIPS16 mode using the -mips16 compiler option.

Preprocessor definitions

-D_RSDK_|_RSDK_v15_|__RSDK__|__RSDK__v15__

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Define identifier for toolchain
Dependency: None
Status: New
Version: 1.5.0+
Description:

This identifier provides preprocessor identification for toolchain and toolchain version. For users who need to tune the source code between toolchain (such as MSDK and RSDK) and toolchain version (such as RSDK-1.3 and RSDK-1.5).

-D__m3081|__m4081|__m4180|__m4181|__m4281|__m5181|__m5280|__m5281

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Define identifier for each processor
Dependency: None
Status: Current
Version: 1.5.0+
Description:

Due to differences among the ISAs of Realtek's processor cores, users may need fine-tune certain code segments for each core, e.g. fine-tune machine-dependent codes using inline assembly. When the target processor is set by specifying -march|-mtune|-mcpu, compiler will automatically add the corresponding preprocessor identifier for users to separate machine-dependent codes in the same segment. The preprocessor identifiers are shown in table 2.5.

Table 2.5: Preprocessor definition mapping

Processor Core	Preprocessor definition
-march=3081	-D__m3081
-march=4081	-D__m4081
-march=4180	-D__m4180
-march=4181	-D__m4181
-march=4281	-D__m4281
-march=5181	-D__m5181
-march=5280	-D__m5280
-march=5281	-D__m5281

Program 3: Example of using preprocessor definitions

```
#ifdef __m4180
  machine-dependent code for 4180
#endif

#ifdef __m4181
  machine-dependent code for 4181
#endif

#ifdef __m5181
  machine-dependent code for 5181
#endif

#ifdef __m5280
  machine-dependent code for 5280
#endif
```

-D__rlx_gprlock|__rlx_no_gprlock

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Define identifier for hardware-interlock support
Dependency: None
Status: Current
Version: 1.5.7+
Description:

This identifier is used to checking the specified processor has hardware-interlock support or not. For the processors without hardware-interlock support, users have to note the load-use issue when writing assembly. The preprocessor identifiers are shown in table 2.6.

Table 2.6: Preprocessor definition for hardware interlock

Processor Core	Preprocessor definition
-march=3081|4081|4180|4181|5181	-D__rlx_no_gprlock
-march=4281|5280|5281	-D__rlx_gprlock

Program 4: Example of using interlock preprocessor definitions

```
foo:
        lw      $4, 0x0($sp)
#ifdef __rlx_gprlock
        nop
#endif
        addu    $3, $4, $5
```

-D__USE_ULS__|__FIX_BDSL__

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Define identifier for compilation option
Dependency: None
Status: New
Version: 1.5.7+
Description:

These identifier are used to checking a compilation option for a hardware feature is enabled or disabled. Now we support preprocessor identifications for compilation option ‘-fuse-uls’ and ‘-ffix-bdsl’.

Chapter 3 Binutils

The binutils tool set is based on the GNU binutils package. The binutils tool set includes assembler, linker, and object file manipulation utilities, such as ar, nm, objdump, and objcopy. In RSDK 4.6.4 , the binutils has been upgraded from version 2.19 to 2.22 to provide a more reliable and a more stable development environment for both MIPS1 and MIPS16 applications.

The list of changes is summarized in the following subsections.

Assembler

-march=3081|4081|4180|4181|4281|5181|5280|5281

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Set the target processor core
Status: Current
Description:

The -march option sets the target processor core to the one specified in the option.
If none of the -march option is specified, the assembler will automatically add -march=4180 to the option list.

MIPS1 and MIPS16 ISA mode

RSDK assembler (GNU AS) supports writing assembly in mips1 and mips16 ISA-mode: the directive `.set mips16` puts the assembler into MIPS 16 mode, and use `.set nomips16` to return to normal 32 bit mode.

RSDK linker (GNU LD) can automatically link mixed mips1 and mips16 objects and automatically translate jump instructions to fit the ISA-mode of target symbols (such as `jal` → `jalx`). Here list the jump instructions that support cross ISA-mode jump:

- `jr and jalr`: the LSB of jump register is used to determine the ISA mode (0: MIPS1, 1: MIPS16), and the value of jump register are set by linker (symbol relocation).
- `jalx/jal`: `jal` is automatically translated to `jalx` when caller and callee belong to different ISA-modes.

NOTE: other branch and jump instruction (such as `j` and `bal`) don't support cross ISA-mode jumps and linker will raise error for that.

Load-Use Detection

Load-use detection is enabled automatically for the processors without hardware-interlock (RLX4081, LX4180, RLX4181, and RLX5181) For a reorder section, the assembler inserts appropriate `nop` to avoid using a regsiter before finishing loading its value. For a nonreorder section, the assembler outputs warning for the inappropriate load-use instructions.

Program 5: Example of load-use detection

```
------ file test.s --------
        .set noreorder
foo:
        lw      $4, 0x0($sp)
        addu    $3, $4, $5
bar:
        nop
---------------------------

% mips-elf-as -march=4181 test.s

test.s: Assembler messages:
test.s:4: Warning: LOAD-USE: regno=4
```

-ffix-bdsl

This option is only for Taroko processors (prior to v1.3) to fix the known issues: Branch-Delay-Slot-Load (BDSL) and immediate boundary problem. When this option is specified, the assembler will do following fix and detection:

- Not put load instruction in branch delay slot at mips1 mode.
- Not put any instruction in branch delay slot at mips16 mode.
- Raises warning to user if BDSL issue occur at noreorder section (assembly).
- Auto-extend mips16 instruction as 32-bit for inappropriate immediate boundary value.

Program 6: Example of BDSL

```
        .set noreorder
foo:
        li      $3, 0x1
        li      $2, 0x1
        li      $4, 0x1
        bne     $3, $2, bar
        lw      $4, 0x0($sp)
bar:
        bne     $4, $2, bar
        addu    $5, $4, $3
        nop
```

-warn-possible-load-use

When this option is specified, the assembler outputs warning for the load instruction at branch delay slot. For the processors without hardware-interlock support, a load instruction at branch delay slot, it may be used by the next instruction at the target address. Therefore these load instructions have potential load-use issue.

-warn-missing-delay-slot

When this option is specified, the assembler outputs warning for the load instruction at the end of assembly fragment (such as the end of section, before a new directive). This means, check the missing instruction for delay slot for user.

Program 7: A potential load-use case

```
        .set noreorder
foo:
        ...
        bne     $3, $2, bar
        lw      $4, 0x0($sp)
        ...
bar:
        addu    $5, $4, $3
        nop
```

Program 8: An example of missing delay-slot

```
        .set noreorder
foo:
        ...
        bne     $3, $2, bar
bar:
        addu    $5, $4, $3
```

Objdump

-mmips:3081|4081|4180|4181|4281|5181|5280|5281

Architecture: RX3081, RLX4081, LX4180, RLX4181, RX4281, RLX5181, LX5280, RX5281
Summary: Set the target processor core
Status: Current
Description:

The -mmips option specifies the target ISA for the objdump utility. The default ISA is set as RX5281. The usage is shown in program 9.

Program 9: Objdump Example

```
rsdk-elf-objdump -d -mmips:3081 file.o
rsdk-elf-objdump -Dr file.o
rsdk-elf-objdump -DR share.so
```

Run-Time OPCODE Table

In RSDK 4.6.4 , binutils has been patched to support dynamic opcode tables. This is done by storing the opcode table in an external file and by loading the external file during run-time. The Run-Time opcode table mechanism enables a single toolchain to support multiple instruction sets. This feature is useful for projects with custom engines and user defined instructions.

An example of opcode table is shown as follows:

```
/* These instructions appear first so that the disassembler will find
   them first.  The assemblers uses a hash table based on the
   instruction name anyhow.  */
  name,    args,      match,      mask,       pinfo,      pinfo2,     membership
```

```
{"pref",   "k,o(b)",  0xcc000000, 0xfc000000, RD_b,       0,          I4|I32|G3},
{"prefx",  "h,t(b)",  0x4c00000f, 0xfc0007ff, RD_b|RD_t,  0,          I4|I33},
{"nop",    "",        0x00000000, 0xffffffff, 0,          INSN2_ALIAS,I1}
```

Instruction Fields

Each instruction in the opcode table contains seven arguments. They are **name**, **args**, **match**, **mask**, **pinfo**, **pinfo2**, and **membership**. These seven arguments define the mnemonic name and the format, as well as information for encoding and decoding for an instruction. The detail for each argument is explained in the following subsection.

- **name:**
  This field is the name of the instruction.
- **args:**
  This field is a string describing the arguments for this instruction.

  These are the characters which may appear in the args field of an instruction. They appear in the order in which the fields appear when the instruction is used. Commas and parentheses in the args string are ignored when assembling, and written into the output when disassembling.

  Each of these characters corresponds to a mask field defined above.

```
   "1" 5 bit sync type (OP_*_SHAMT)
   "<" 5 bit shift amount (OP_*_SHAMT)
   ">" shift amount between 32 and 63, stored after subtracting 32 (OP_*_SHAMT)
   "a" 26 bit target address (OP_*_TARGET)
   "b" 5 bit base register (OP_*_RS)
   "c" 10 bit breakpoint code (OP_*_CODE)
   "d" 5 bit destination register specifier (OP_*_RD)
   "h" 5 bit prefx hint (OP_*_PREFX)
   "i" 16 bit unsigned immediate (OP_*_IMMEDIATE)
   "j" 16 bit signed immediate (OP_*_DELTA)
   "k" 5 bit cache opcode in target register position (OP_*_CACHE)
       Also used for immediate operands in vr5400 vector insns.
   "o" 16 bit signed offset (OP_*_DELTA)
   "p" 16 bit PC relative branch target address (OP_*_DELTA)
   "q" 10 bit extra breakpoint code (OP_*_CODE2)
   "r" 5 bit same register used as both source and target (OP_*_RS)
   "s" 5 bit source register specifier (OP_*_RS)
   "t" 5 bit target register (OP_*_RT)
   "u" 16 bit upper 16 bits of address (OP_*_IMMEDIATE)
   "v" 5 bit same register used as both source and destination (OP_*_RS)
   "w" 5 bit same register used as both target and destination (OP_*_RT)
   "U" 5 bit same destination register in both OP_*_RD and OP_*_RT
       (used by clo and clz)
   "C" 25 bit coprocessor function code (OP_*_COPZ)
   "B" 20 bit syscall/breakpoint function code (OP_*_CODE20)
   "J" 19 bit wait function code (OP_*_CODE19)
   "x" accept and ignore register name
   "z" must be zero register
   "K" 5 bit Hardware Register (rdhwr instruction) (OP_*_RD)
   "+A" 5 bit ins/ext/dins/dext/dinsm/dextm position, which becomes
        LSB (OP_*_SHAMT).
Enforces: 0 <= pos < 32.
   "+B" 5 bit ins/dins size, which becomes MSB (OP_*_INSMSB).
Requires that "+A" or "+E" occur first to set position.
Enforces: 0 < (pos+size) <= 32.
   "+C" 5 bit ext/dext size, which becomes MSBD (OP_*_EXTMSBD).
Requires that "+A" or "+E" occur first to set position.
Enforces: 0 < (pos+size) <= 32.
```

```
(Also used by "dext" w/ different limits, but limits for
that are checked by the M_DEXT macro.)
   "+E" 5 bit dinsu/dextu position, which becomes LSB-32 (OP_*_SHAMT).
Enforces: 32 <= pos < 64.
   "+F" 5 bit "dinsm/dinsu" size, which becomes MSB-32 (OP_*_INSMSB).
Requires that "+A" or "+E" occur first to set position.
Enforces: 32 < (pos+size) <= 64.
   "+G" 5 bit "dextm" size, which becomes MSBD-32 (OP_*_EXTMSBD).
Requires that "+A" or "+E" occur first to set position.
Enforces: 32 < (pos+size) <= 64.
   "+H" 5 bit "dextu" size, which becomes MSBD (OP_*_EXTMSBD).
Requires that "+A" or "+E" occur first to set position.
Enforces: 32 < (pos+size) <= 64.

   Floating point instructions:
   "D" 5 bit destination register (OP_*_FD)
   "M" 3 bit compare condition code (OP_*_CCC) (only used for mips4 and up)
   "N" 3 bit branch condition code (OP_*_BCC) (only used for mips4 and up)
   "S" 5 bit fs source 1 register (OP_*_FS)
   "T" 5 bit ft source 2 register (OP_*_FT)
   "R" 5 bit fr source 3 register (OP_*_FR)
   "V" 5 bit same register used as floating source and destination (OP_*_FS)
   "W" 5 bit same register used as floating target and destination (OP_*_FT)

   Coprocessor instructions:
   "E" 5 bit target register (OP_*_RT)
   "G" 5 bit destination register (OP_*_RD)
   "H" 3 bit sel field for (d)mtc* and (d)mfc* (OP_*_SEL)
   "P" 5 bit performance-monitor register (OP_*_PERFREG)
   "e" 5 bit vector register byte specifier (OP_*_VECBYTE)
   "%" 3 bit immediate vr5400 vector alignment operand (OP_*_VECALIGN)
   see also "k" above
   "+D" Combined destination register ("G") and sel ("H") for CP0 ops,
for pretty-printing in disassembly only.

   Macro instructions:
   "A" General 32 bit expression
   "I" 32 bit immediate (value placed in imm_expr).
   "+I" 32 bit immediate (value placed in imm2_expr).
   "F" 64 bit floating point constant in .rdata
   "L" 64 bit floating point constant in .lit8
   "f" 32 bit floating point constant
   "l" 32 bit floating point constant in .lit4

   MDMX instruction operands (note that while these use the FP register
   fields, they accept both $fN and $vN names for the registers):
   "O" MDMX alignment offset (OP_*_ALN)
   "Q" MDMX vector/scalar/immediate source (OP_*_VSEL and OP_*_FT)
   "X" MDMX destination register (OP_*_FD)
   "Y" MDMX source register (OP_*_FS)
   "Z" MDMX source register (OP_*_FT)

   DSP ASE usage:
   "2" 2 bit unsigned immediate for byte align (OP_*_BP)
   "3" 3 bit unsigned immediate (OP_*_SA3)
   "4" 4 bit unsigned immediate (OP_*_SA4)
   "5" 8 bit unsigned immediate (OP_*_IMM8)
   "6" 5 bit unsigned immediate (OP_*_RS)
   "7" 2 bit dsp accumulator register (OP_*_DSPACC)
   "8" 6 bit unsigned immediate (OP_*_WRDSP)
```

```
"9" 2 bit dsp accumulator register (OP_*_DSPACC_S)
"0" 6 bit signed immediate (OP_*_DSPSFT)
":" 7 bit signed immediate (OP_*_DSPSFT_7)
"'" 6 bit unsigned immediate (OP_*_RDDSP)
"@" 10 bit signed immediate (OP_*_IMM10)

MT ASE usage:
"!" 1 bit usermode flag (OP_*_MT_U)
"$" 1 bit load high flag (OP_*_MT_H)
"*" 2 bit dsp/smartmips accumulator register (OP_*_MTACC_T)
"&" 2 bit dsp/smartmips accumulator register (OP_*_MTACC_D)
"g" 5 bit coprocessor 1 and 2 destination register (OP_*_RD)
"+t" 5 bit coprocessor 0 destination register (OP_*_RT)
"+T" 5 bit coprocessor 0 destination register (OP_*_RT) - disassembly only

Other:
"()" parens surrounding optional value
","  separates operands
"[]" brackets around index for vector-op scalar operand specifier (vr5400)
"+"  Start of extension sequence.

Characters used so far, for quick reference when adding more:
"1234567890"
"%[]<>(),+:'@!$*&"
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
"abcdefghijklopqrstuvwxz"

Extension character sequences used so far ("+" followed by the
following), for quick reference when adding more:
"1234"
"ABCDEFGHIPQSTX"
"pstx"
```

- **match:**
  The basic opcode for the instruction. When assembling, this opcode is modified by the arguments to produce the actual opcode that is used. If pinfo is INSN_MACRO, then this is 0.

- **mask:**
  If pinfo is not INSN_MACRO, then this is a bit mask for the relevant portions of the opcode when disassembling. If the actual opcode anded with the match field equals the opcode field, then we have found the correct instruction. If pinfo is INSN_MACRO, then this fiend is the macro identifier.

- **pinfo:**
  For a macro, this is INSN_MACRO. Otherwise, it is a collection of bits describing the instruction, notably any relevant hazard information.

  These are the bits which may be set in the pinfo field of an instructions, if it is not equal to INSN_MACRO.

```
WR_d: /* Modifies the general purpose register in OP_*_RD.  */
WR_t: /* Modifies the general purpose register in OP_*_RT.  */
WR_31: /* Modifies general purpose register 31.  */
WR_D: /* Modifies the floating point register in OP_*_FD.  */
WR_S: /* Modifies the floating point register in OP_*_FS.  */
WR_T: /* Modifies the floating point register in OP_*_FT.  */
RD_s: /* Reads the general purpose register in OP_*_RS.  */
RD_t: /* Reads the general purpose register in OP_*_RT.  */
RD_S: /* Reads the floating point register in OP_*_FS.  */
RD_T: /* Reads the floating point register in OP_*_FT.  */
RD_R: /* Reads the floating point register in OP_*_FR.  */
WR_CC: /* Modifies coprocessor condition code.  */
RD_CC: /* Reads coprocessor condition code.  */
```

```
/* TLB operation.  */
#define INSN_TLB                    0x00002000
RD_C0: /* Reads coprocessor register other than floating point register.  */
RD_C1: /* Reads coprocessor register other than floating point register.  */
RD_C2: /* Reads coprocessor register other than floating point register.  */
RD_C3: /* Reads coprocessor register other than floating point register.  */

/* Instruction loads value from memory, requiring delay.  */
#define INSN_LOAD_MEMORY_DELAY      0x00008000

/* Instruction loads value from coprocessor, requiring delay.  */
#define INSN_LOAD_COPROC_DELAY     0x00010000
/* Instruction has unconditional branch delay slot.  */
#define INSN_UNCOND_BRANCH_DELAY    0x00020000
/* Instruction has conditional branch delay slot.  */
#define INSN_COND_BRANCH_DELAY      0x00040000
/* Conditional branch likely: if branch not taken, insn nullified.  */
#define INSN_COND_BRANCH_LIKELY     0x00080000
/* Moves to coprocessor register, requiring delay.  */
#define INSN_COPROC_MOVE_DELAY      0x00100000
/* Loads coprocessor register from memory, requiring delay.  */
#define INSN_COPROC_MEMORY_DELAY    0x00200000
/* Reads the HI register.  */
#define INSN_READ_HI     0x00400000
/* Reads the LO register.  */
#define INSN_READ_LO     0x00800000
/* Modifies the HI register.  */
#define INSN_WRITE_HI     0x01000000
/* Modifies the LO register.  */
#define INSN_WRITE_LO     0x02000000
/* Takes a trap (easier to keep out of delay slot).  */
#define INSN_TRAP                   0x04000000
/* Instruction stores value into memory.  */
#define INSN_STORE_MEMORY     0x08000000
/* Instruction uses single precision floating point.  */
#define FP_S     0x10000000
/* Instruction uses double precision floating point.  */
#define FP_D     0x20000000
/* Instruction is part of the tx39's integer multiply family.    */
#define INSN_MULT                   0x40000000
/* Instruction synchronize shared memory.  */
#define INSN_SYNC     0x80000000

/* These are the bits which may be set in the pinfo2 field of an
   instruction. */

/* Instruction is a simple alias (I.E. "move" for daddu/addu/or) */
#define INSN2_ALIAS     0x00000001
/* Instruction reads MDMX accumulator. */
#define INSN2_READ_MDMX_ACC     0x00000002
/* Instruction writes MDMX accumulator. */
#define INSN2_WRITE_MDMX_ACC     0x00000004

/* UDI instruction */
#define INSN2_UDI                   0x00100000
/* DMP CEI - identify an opcode is DMP CEI or not. */
#define INSN2_DMP_CEI               0x00200000
#define INSN2_TWORD_LOAD            0x00400000
#define INSN2_TWORD_USE             0x00800000
```

```
#define INSN2_EXTEND_ONLY          0x01000000

/* Instruction is actually a macro.  It should be ignored by the
   disassembler, and requires special treatment by the assembler.  */
#define INSN_MACRO                 0xffffffff

/* Masks used to mark instructions to indicate which MIPS ISA level
   they were introduced in.  ISAs, as defined below, are logical
   ORs of these bits, indicating that they support the instructions
   defined at the given level.  */
```

- **pinfo2:**
  A collection of additional bits describing the instruction.
- **membership:**
  A collection of bits describing the instruction sets of which this instruction or macro is a member.

  For RLX processor cores, the following ISA sets are defined:

  – ISA_RLX4180
  – ISA_RLX4181
  – ISA_RLX5181
  – ISA_RLX5280

Custom UDI Instructions

To add custom UDI instructions to the binutils, a text-based UDI instruction table must be constructed. This UDI instruction table is used to encode assembly code and to decode between machine object code.

UDI Instruction File

Example UDI instruction file is shown as follows:

```
/*
 * RLX UDI instruction list.
 */
struct mips_opcode rlx_udi_opcodes[] =
{
/* Lexra opcode extensions. Register mode */
{"udi0",        "d,v,t",       0x00000038, 0xfc0007ff, WR_d|RD_s|RD_t,INSN2_UDI,RUDI
},
{"udi1",        "d,v,t",       0x0000003a, 0xfc0007ff, WR_d|RD_s|RD_t,INSN2_UDI,RUDI
},
{"udi2",        "d,v,t",       0x0000003b, 0xfc0007ff, WR_d|RD_s|RD_t,INSN2_UDI,RUDI
},
{"udi3",        "d,v,t",       0x0000003c, 0xfc0007ff, WR_d|RD_s|RD_t,INSN2_UDI,RUDI
},
{"udi4",        "d,v,t",       0x0000003e, 0xfc0007ff, WR_d|RD_s|RD_t,INSN2_UDI,RUDI
},
{"udi5",        "d,v,t",       0x0000003f, 0xfc0007ff, WR_d|RD_s|RD_t,INSN2_UDI,RUDI
},

/* Lexra opcode extensions. Immediate mode */
{"udi0i",       "t,r,j",       0x60000000, 0xfc000000, WR_t|RD_s,INSN2_UDI,RUDI
},
{"udi1i",       "t,r,j",       0x64000000, 0xfc000000, WR_t|RD_s,INSN2_UDI,RUDI
},
{"udi2i",       "t,r,j",       0x68000000, 0xfc000000, WR_t|RD_s,INSN2_UDI,RUDI
},
```

```
{"udi3i",       "t,r,j",       0x6c000000, 0xfc000000, WR_t|RD_s,INSN2_UDI,RUDI
}
};
```

Instruction File Manipulation

The manipulation of ISA file is done via a utility program, rsdk-elf-opcutil, which is shipped with the RSDK toolchain package.

The usage of rsdk-elf-opcutil is shown as follows:

```
sh% ./rsdk-elf-opcutil

RLX Binutils OPCODE Util v1.4

usage: ./rsdk-elf-opcutil [-hlv] [-i [isa.bin]] [-r isa.bin]
        -h: help
        -l: list opcode table tags
        -d: show ISA info of the default file
        -i: show ISA info of the file
        -r: replace opcode table
        -v: verbose output
```

- **-l**: the -l option lists the current supported opcode tables. The output includes name, description, number of ISA, and number of UDI instructions of each opcode table. Example is shown as follows:

  ```
  sh% ./rsdk-elf-opcutil -l

  RLX Binutils OPCODE Util v1.5

  optree[0] = RLX, RLX opcode v1.5 rev 2, num_isa = 1732, num_rlx = 273, num_udi = 10
  optree[1] = VENUS, DVR-VENUS opcode v1.5 rev 3, num_isa = 1732, num_rlx = 273, num_ud
  optree[2] = MARS, DVR-MARS opcode v1.5 rev 3, num_isa = 1732, num_rlx = 273, num_udi
  optree[3] = JUPITER, DVR-JUPITER opcode v1.5 rev 10 (20091207), num_isa = 1732, num_r
  optree[4] = SATURN, DVR-SATURN opcode v1.5 rev 10 (20100129), num_isa = 1732, num_rlx
  optree[5] = DARWIN, TV-DARWIN opcode v1.5 rev 10 (20100601), num_isa = 1732, num_rlx
  optree[6] = MERCURY, MERCURY opcode v1.5 rev 10 (20101109), num_isa = 1732, num_rlx =
  optree[7] = NIKE, DVR-NIKE opcode v1.5 rev 10 (20111220), num_isa = 1732, num_rlx = 2
  optree[8] = MAGELLAN, TV-MAGELLAN opcode v1.5 rev 10 (20120704), num_isa = 1732, num_
  ```

  Note that some instructions not support by RLX, LX, and RX processors will be automatically stripped by genopc (such as instruction `abs.ps` for Loongson 2), so we can actually get smaller opcode table than opcutil displays.

- **-d**: the -d option shows details of the current opcode table. The output includes the file name, tag, number of ISA, and number of UDI instructions of the opcode table. Example is shown as follows:

  ```
  sh% ./rsdk-elf-opcutil -d

  RLX Binutils OPCODE Util v1.5

  Filename: ../mips-elf/bin/rlxisa.bin
  TAG: RLX opcode v1.5 rev 2
  ISA: 1060 instructions
  UDI: 10 instructions
  ```

- **-i**: the -i option shows details of the specified opcode file. The output includes the file name, tag, number of ISA, and number of UDI instructions of the opcode table. Example is shown as follows:

```
sh% ./rsdk-elf-opcutil -i rlxisa-mars.bin

Filename: rlxisa-mars.bin
TAG: DVR-MARS opcode v1.5 rev 3
ISA: 1070 instructions
UDI: 252 instructions
```

- **-r**: the -r option replaces the current opcode table file with the specified opcode tag. Users can use -l option to find out the supported opcode tags.

```
sh% ./rsdk-elf-opcutil -r VENUS

RLX Binutils OPCODE Util v1.5

Changing default opcode table to VENUS ...
Filename: rlxisa.bin
TAG: DVR-VENUS opcode v1.5 rev 3
ISA: 1070 instructions
UDI: 89 instructions
```

Chapter 4 GDB

GDB is the GNU project debugger, The purpose is to allow you to see what is going on "inside" another program while it executes or what another program was doing at the moment it crashed.

In RSDK 4.6.4 , gdb has been upgraded to 7.2.91 for performance improvement and bug fixes. In addition, some special functions been added for it. The new functions are summarized in the following subsections.

Register Groups

We add new commands to show In-house processor's registers in groups, the register groups include: general, float, system, cp0, lxcp0, cp3, radiax. The related commands are:

- info registers general.
- info registers float.
- info registers system.
- info registers cp0.
- info registers lxcp0.
- info registers cp3.
- info registers radiax.

Show Register Detail

Some configure registers have complex bit-filed function design, but GDB can only give the value of the register. So we want to print the detail information of a register's bit-fileds. No new command added, the detail information be printed by "info register $register_name":

```
  . . . .
(gdb)  i r \$pc
pc: 0x80001168
(gdb)  i r \$Status
Status: 0x20000000
 [31-28] [27-23] [22] [21-16]  [15-8] [7-6] [5] [4] [3] [2] [1] [0]
 CU[3:0]     ---  BEV     --- IM[7:0]   --- KUo IEo KUp IEp KUc IEc
       2       0    0       0       0     0   0   0   0   0   0   0
(gdb)  i r \$EntryHi
EntryHi: 0x80000000
 [31-12] [11-6] [5-0]
     VPN   ASID   ---
   80000      0     0
  . . . .
```

Record And Repaly For RLX-VM

When GDB work with RLX-VM, some commands supplied if RLX-VM can be recorded and replayed. In this case, GDB only supply user commands, the mainly function is implemented by the record/replay server and OS kernel module. For more detail information about that, please refer to chapter 8 of RLX-VM's UserGuide: Record and replay. The related commands are listed below:

enable rlxvm-record host:port

Enable record and replay function. The target of GDB must be remote when doing this. GDB will connect to record server and create a save point at the first stop point of the program being debugged. In addition to the first save point, user can use GDB commands to create save points when GDB stops, or tell GDB to create save points automatically for every special time interval if the program needs to run a long time before stopping. We cannot create duplicated save points at one simulate time of RLX-VM.

show rlxvm-record-status

Show if recording function is enable.

disable rlxvm-record

Disable record and replay function. GDB will disconnect from record server.

set rlxvm-recordtime-interval 'seconds'

Set the time interval to create save point automatically when the program is running, default value is 5s (host time).

show rlxvm-recordtime-interval

Show the time interval to create save point automatically when the program is running.

set rlxvm-recordfile-limit 'size_limit'

Set total size of save points(in MB). GDB checks it when a new save point be created, if the total size of save points is bigger than 'size_limit', the second one (accord to the simulate time of RLX-VM) will be deleted. The unit is not needed when input size_limit, default value is 500(M).

show rlxvm-recordfile-limit

Show the total size limitation of save points.

rlxvm-save (vs)

Create a save point.

rlxvm-replay (vr) 'save_point_index'

Replay rlxvm's status and run from the save point distinguished by the 'save_point_index'. RLX-VM will be resumed to the next instruction after the save point.

info rlxvm-savepoint

Print the names of all the save points, which are sort by simulation time.

delete rlxvm-savepoint(d s) 'save_point_index'

Delete a special save point assigned by 'save_point_index', if no index assigned, GDB will delete all thesave points after confirm with user.

rlxvm-savepoint-location 'index'

Show the location of save point(s), this command can show the full path of save point's source file.

How To Replay/Record RLX-VM

This section describes how to use record and replay function of RLX-VM. It is only supported for Linux OS:

- RedHat Enterprise 4.7 (kernel: 2.6.9-78, built with GCC: 3.4.6)
- RedHat Enterprise 5.6 (kernel: 2.6.18-238, built with GCC: 4.1.2)

In order to support this function, we have developed a record server, a Linux kernel module (rlxvm_record_module.ko), The record server (record_server) will be connected to GDB through TCP socket, It receives GDB commands, and controls the Linux kernel module to store/resume the status of RLX-VM to save points. RLX-VM, rlxvm_record_module.ko and record_server must run on the same host. Before record and replay RLX-VM, Linux record kernel module must be installed with root privilege. The Linux record kernel module (rlxvm_record_module.ko) and installing script (rlxvm_record_module.sh) can be found in INSTALL_DIR/bin/OS_kernel, user can use the following commands to install and uninstall Linux record kernel module:

- rlxvm_record_module.sh install
- rlxvm_record_module.sh uninstall

The procedures to use RLX-VM with record and replay:

- Start RLX-VM with GDB support
- Start record server: record_server :tcp_port
- Start GDB
- Select a program for GDB to debug: (gdb) file program
- Connect to RLX-VM: (gdb) target remote host: gdb_port
- Connect to record server: (gdb) enable rlxvm-record host: record_port
- Do normal debugging
- Create snapshots of RLX-VM, GDB will give a point index: (gdb) rlxvm-save
- Replay RLX-VM to a special snapshot: (gdb) rlxvm-replay point_index

Chapter 5 Problem Report

The official website for the processor and platform team is at the following URL:

http://processor.realtek.com.tw

On the official website, latest news, documentation, and releases of RSDK toolchain will be made available as soon as they are ready. The link to the issue tracking system can also be found on the processor website. Through the issue tracking system, any feature request and bug report will be handled in a systematic and timely fashion.

To report a problem, in addition to the detail problem description, please also clearly indicate the platform, the RSDK version, and exact way to reproduce the problem. The more details we have, the faster we can have the problem identified and nailed.

Appendix A RADIAX registers

RADIAX register name translation

For processor cores that support DSP instructions, an additional set of registers are available for programming. The mnemonic names of RADIAX registers are added to: **${RSDK}/include/regdef.h**

The set of registers are shown as follows:

```
#define m0l         $1
#define m0h         $2
#define m0          $3
#define m1l         $5
#define m1h         $6
#define m1          $7
#define m2l         $9
#define m2h         $10
#define m2          $11
#define m3l         $13
#define m3h         $14
#define m3          $15
#define estatus     $0
#define ecause      $1
#define intvec      $2
#define cbs0        $0
#define cbs1        $1
#define cbs2        $2
#define cbe0        $4
#define cbe1        $5
#define cbe2        $6
#define lps0        $16
#define lpe0        $17
#define lpc0        $18
#define mmd         $24
```

The RADIAX registers are treated as regular MIPS registers in the way the register name translation is processed.

The register name translation can happen in two places:

- **preprocessor:**
  If preprocessor is applicable and the **regdef.h** header file is included, the preprocessor can do the following translation:

  addma.s m0l, m0l, m0l => addma.s $1, $1, $1

- **assembler:**
  When it comes to the assembler, the register names should be prefixed with a $ character. For example, the

assembler is able to do the translation of the following form:

addma.s $m0l, $m0l, $m0l => addma.s $1, $1, $1

Appendix B Inline Assembly Format

Form 1

The first form of inline assembly is shown as follows:

```
asm("move $6, $8");
```

The above code will be translated literally to the code segment shown below:

```
#APP
move $6, $8
#NO_APP
```

Form 2

The second form of inline assembly is shown as follows:

```
int v1;
int v2;

asm("move %0,%1" : "=d"(v1) : "d"(v2));
```

The above inline assembly code states that: copy the value of variable v2 to variable v1 while storing v1 and v2 in general registers. The compiler will generate the actual assembly code as follows:

```
#APP
move $6, $8
#NO_APP
```

NOTE: the actual register number is determined by compiler during register allocation so the actual register number might be different from the one shown above.

Form 3

The third form of inline assembly is shown as follows:

```
register int v1 asm("8");
int ret;

ret = ret + v1;
```

The compiler supports a special keyword, asm, for variable declaration. The above code forces compiler to allocate register 8 for variable v1. The translated assembly code is shown as follows:

```
#APP
addu   $8, $2, $8
#NO_APP
```

There are five alternatives to the above form. The $8, %8, and #8 are internally supported in the compiler. The name stands for the alias of the register.

```
register int v1 asm("8");
register int v1 asm("$8");
register int v1 asm("%8");
register int v1 asm("#8");
register int v1 asm("name");
```

NOTE: The compiler optimization, -O, has the priority over register number assignment in this form. If -O is turned on, the compiler might assign a different register number than the one specified in the inline assembly code.

Appendix C RELEASE NOTE

```
----------------
RSDK Release 4.6
----------------

We are pleased to announce the release of RSDK version 4.6 on September 24, 2012.
RSDK stands for Realtek Software Development Kit. It is the software development
kit for Realtek's in-house processor cores. Version 4.6.3 is the first stable
release for branch 4.6.

What's new in release 4.6:

1. gcc-4.6.3

The major version of gcc has come to 4.6. At this moment the note is made,
it reached 4.6.3 with less bugs and more stability. Here is a brief list of
 notable technique items that are added or enhanced since 4.4.

  4.5) GCC has been integrated with the MPC library. This allows GCC to
       evaluate complex arithmetic at compile time more accurately.

  4.5) A new link-time optimizer has been added (-flto). When this option
       is used, GCC generates a bytecode representation of each input file
       and writes it to specially-named sections in each object file.
       When the object files are linked together, all the function bodies
       are read from these named sections and instantiated as if they had
       been part of the same translation unit. This enables interprocedural
       optimizations to work across different files (and even different
       languages), potentially improving the performance of the generated
       code.

  4.5) MIPS: GNU/Linux targets can now generate read-only .eh_frame sections.
       This optimization requires GNU binutils 2.20 or above, and is only
       available if GCC is configured with a suitable version of binutils.

  4.5) MIPS: GNU/Linux targets can now attach special relocations to indirect
       calls, so that the linker can turn them into direct jumps or branches.

  4.5) MIPS: GCC supports four new function attributes for interrupt handlers:
       interrupt, use_shadow_register_set, keep_interrupts_masked and
       use_debug_exception_return.

  4.6) GCC now has stricter checks for invalid command-line options, such as
       --as-needed are now rejected and -Wl,--as-needed should be used.

  4.6) The C-only intermodule optimization framework (IMA, enabled by -combine)
       has been removed in favor of the new generic link-time optimization
       framework (LTO) introduced in GCC 4.5.0.

  4.6) New -Wunused-but-set-variable and -Wunused-but-set-parameter warnings
```

were added for C, C++, Objective-C and Objective-C++. These warnings diagnose variables respective parameters which are only set in the code and never otherwise used.

4.6) A new switch -fstack-usage has been added. It makes the compiler output stack usage information for the program, on a per-function basis, in an auxiliary file.

4.6) A new warning, enabled by -Wdouble-promotion, has been added that warns about cases where a value of type float is implicitly promoted to double. This is especially helpful for CPUs that handle the former in hardware, but emulate the latter in software.

2. RSDK Supplementary Library Module

A supplementary library module has been added to enrich RSDK’s capability in functional profiling, performance tuning, and remote debugging. The supplementary library includes following modules:

a. CP3 library - CP3 performance counter
b. Profiler library - function-level profiling support
c. GDB I/O - remote I/O via GDB remote serial protocol
d. RLXCOV - RLX code coverage analysis library
e. RLXULS - RLX unaligned load/store library

CPUs supported by RSDK release 4.6

1. RX3081:
All versions

3. RLX4081:
All versions

3. LX4180:
All versions

4. RLX4181
All versions

5. LX5280
All versions

6. RLX5181
All versions

7. RLX4281
All versions

8. RLX5281
All versions

Appendix D Change Log

```
version 4.6.4

    * Upgrade gcc to 4.6.4
    * Upgrade gdb to 7.6.1
    * RSDK: gcc: enable data-in-code for 3081

version 4.6.3

    * Upgrade to bintutils-2.22, gcc-4.6.3, newlib-2.21
    * Support Link-Time Optimization (LTO)
    * Upgrade binutils to 2.23
    * Remove _ecount
```