GNU Toolchain for ARC Documentation

version 2020.09

Synopsys
October 20, 2020

Contents

Building baremetal applications

Hostlink and libgloss

Newlib project is split into two parts: newlib and libgloss. Newlib implements system-agnostic function, like string formatting used in `vprintf()` or in math library, while libgloss implements system-specific functions, like `_read()` or `_open()` which strongly depend on specific of the execution platform. For example in the baremetal system function `_write()` would work only for stdout and stderr and would write all of the output directly to UART, while in case of a user-space application on top of complete OS, `_write()` would make a system call, and would let OS handle output operation. For this reason, newlib provides an architecture-level library - once compiled it can be used for all compatible ARC-processors, but libgloss would also contain parts specific to the particular "board" be it an ASIC, or FPGA, or a simulator. Usually it is not enough to use just newlib to build a complete application, because newlib doesn't provide even a dummy implementation of `_exit()` function, hence even in the case of simplest applications that do not use system IO, there is still a need to provide implementation of `_exit()` either through linking with some libgloss implementation or by implementing it right in the application.

Because libgloss implementation is often specific to a particular chip (nee BSP), Synopsys provides only two libgloss implementation as of now - libnsim, that supports nSIM GNU IO Hostlink and libnosys, which is really an architecture-agnostic implementation, that simply provides empty stubs for a few of the most used system functions - this is enough to link an application, however it may not function as expected.

nSIM GNU IO Hostlink works via software exceptions, just like the syscalls in real OS - when target application needs something to be done by the hostlink, it causes a software exception with parameters that specify what action is required. nSIM intercepts those exceptions and handles them. The advantage of this approach is that same application binary can be used with other execution environments, which also handle software exceptions - unlike the case where a system function implementation is really baked inside the application binary.

To use hostlink in application, add option `--specs=nsim.specs` to gcc options when linking - library libnsim.a will be linked in and will provide implementations to those system level functions

To use a generic libnosys library, add option `--specs=nosys.specs` to gcc options when linking. Note that one of the important distinction between libnsim and libnosys is that `_exit()` implementation in libnosys is an infinite loop, while in libnsim it will halt the CPU core. As a result, at the end of an execution, application with libnosys will spin, while application with libnsim will halt.

If you are a chip and/or OS developer it is likely that you would want to provide a libgloss implementation appropriate for your case, because libnsim.a is not intended to be in the real-world production baremetal applications.

Understanding GCC -mcpu option

The GCC option `-mcpu=` for ARC does not only designate the ARC CPU family (ARC EM, HS, 600 or 700), but also enables the corresponding set of optional instructions defined for the selected configuration. Therefore a particular `-mcpu` value selects not only a family, but also other `-m<something>` GCC options for ARC.

Value of `-mcpu=` option not only sets various other `-m<something>` options to particular values, but also selects a specific standard library build which was done for this particular core configuration. It is possible to override selection of hardware extensions by passing individual `-m<something>` options to compiler *after* the `-mcpu=` option, however standard library build used for linkage still will be the one matching `-mcpu=` value. Therefore, for example, option combination `-mcpu=em4 -mno-code-density` will generate code that doesn't use code density instructions, however it will be linked with standard library that has been built with just `-mcpu=em4`, which uses code density instructions - therefore final application still may use code density instructions. That's why TCF generator, for example, analyzes hardware features present in the configured processor and selects `-mcpu=` value that is the best match for this configuration.

ARC EM

The following table summarize what options are set by each of the possible `-mcpu` values for ARC EM.

-mcpu values for ARC EM

-mcpu=	-mcode-density	-mnorm	-mswap	-mbarrel-shifter	-mdiv-rem	-mmpy-option	-mfpu	-mrf16
em						none		
em_mini						none		Y
em4	Y					none		
arcem	Y			Y		wlh1		
em4_dmips	Y	Y	Y	Y	Y	wlh1		
em4_fpus	Y	Y	Y	Y	Y	wlh1	fpus	
em4_fpuda	Y	Y	Y	Y	Y	wlh1	fpuda	

The above `-mcpu` values correspond to specific ARC EM Processor templates presented in the ARChitect tool. It should be noted however that some ARC features are not currently supported in the GNU toolchain, for example DSP instruction support, reduced register size and reduced register sets. Relationship between `-mcpu` values above and ARC EM Processor templates in ARChitect tool are limited to options listed in the table. Tables will be updated as support for more options get added to the GNU toolchain.

- `-mcpu=em` doesn't correspond to any specific template, it simply defines the base ARC EM configuration without any optional instructions.
- `-mcpu=em_mini` is same as `em`, but uses reduced register file with only 16 core registers.
- `-mcpu=em4` is a base ARC EM core configuration with `-mcode-density` option. It corresponds to the following ARC EM templates in ARChitect: em4_mini, em4_sensor, em4_ecc, em6_mini, em5d_mini, em5d_mini_v3, em5d_nrg, em7d_nrg, em9d_mini. Note, however, that those `mini` templates has a reduced core register file, while this option doesn't specify it.
- `-mcpu=arcem` doesn't correspond to any specific template, it is legacy flag preserved for compatibility with older GNU toolchain versions, where `-mcpu` used to select only a CPU family, while optional features were enabled or disable by individual `-m<something>` options.
- `-mcpu=em4_dmips` is a full-featured ARC EM configuration for integer operations. It corresponds to the following ARC EM templates in ARChitect: em4_dmips, em4_rtos, em6_dmips, em4_dmips_v3, em4_parity, em6_dmips_v3, em6_gp, em5d_voice_audio, em5d_nrg_v3, em7d_nrg_v3, em7d_voice_audio, em9d_nrg, em9d_voice_audio, em11d_nrg and em11d_voice_audio.
- `-mcpu=em4_fpus` is like `em4_dmips` but with additional support for single-precision floating point unit. It corresponds to the following ARC EM templates in ARChitect: em4_dmips_fpusp, em4_dmips_fpusp_v3, em5d_nrg_fpusp and em9d_nrg_fpusp.
- `-mcpu=em4_fpuda` is like `em4_fpus` but with additional support for double-precision assist instructions. It corresponds to the following ARC EM templates in ARChitect: em4_dmips_fpuspdp and em4_dmips_fpuspdp_v3.
- `-mcpu=quarkse_em` is a configuration for ARC processor in Intel Quark SE chip.

Option	quarkse_em
-mcode-density	Y
-mnorm	Y
-mswap	Y
-mbarrel-shifter	Y
-mdiv-rem	Y

Option	quarkse_em
-mmpy-option	wlh2
-mfpu	quark
-mrf16	

ARC HS

The following table summarize what options are set by each of the possible `-mcpu` values for ARC HS.

-mcpu values for ARC HS

-mcpu=	-mdiv-rem	-matomic	-mll64	-mmpy-option	-mfpu
hs		Y		none	
hs34		Y		mpy	
archs	Y	Y	Y	mpy	
hs38	Y	Y	Y	plus_qmacw	
hs4x	Y	Y	Y	plus_qmacw	
hs4xd	Y	Y	Y	plus_qmacw	
hs38_linux	Y	Y	Y	plus_qmacw	fpud_all

The above `-mcpu` values correspond to specific ARC HS Processor templates presented in the ARChitect tool. It should be noted however that some ARC features are not currently supported in the GNU toolchain, for example reduced register size and reduced register sets. Relationship between `-mcpu` values above and ARC HS Processor templates in ARChitect tool are limited to options listed in the table. Tables will be updated as support for more options get added to the GNU toolchain.

- `-mcpu=hs` corresponds to a basic ARC HS with only atomic instructions enabled. It corresponds to the following ARC HS templates in ARChitect: hs34_base, hs36_base and hs38_base.
- `-mcpu=hs34` is like `hs` but with with additional support for standard hardware multiplier. It corresponds to the following ARC HS templates in ARChitect: hs34, hs36 and hs38.
- `-mcpu=archs` is a generic CPU, which corresponds to the default configuration in older GNU toolchain versions.
- `-mcpu=hs38` is a fully featured ARC HS. It corresponds to the following ARC HS templates in ARChitect: hs38_full
- `-mcpu=hs4x` and `-mcpu=hs4xd` have same option set as `-mcpu=hs38` but compiler will optimize instruction scheduling for specified processors.
- `-mcpu=hs38_linux` is a fully featured ARC HS with additional support for double-precision FPU.

ARC 600 and ARC 700

The following table summarize what options are set by each of the possible `-mcpu` values for ARC 600 and ARC 700.

-mcpu values for ARC 600 and ARC 700

-mcpu	-mnorm	-mswap	-mbarrel-shifter	multiplier
arc700	Y	Y	Y	-mmpy

-mcpu	-mnorm	-mswap	-mbarrel-shifter	multiplier
arc600			Y	
arc600_norm	Y		Y	
arc600_mul64	Y		Y	-mmul64
arc600_mul32x16	Y		Y	-mmul32x16
arc601				
arc601_norm	Y			
arc601_mul64	Y			-mmul64
arc601_mul32x16	Y			-mmul32x16

Linker scripts and memory.x files

Introduction to linker and linker scripts

The way how code and data sections will be organized in the memory by linker strongly depends on the linker script or linker emulation chosen. Linker script (also known as linker command file) is a special file which specifies where to put different sections of ELF file and defines particular symbols which may be used referenced by an application. Linker emulation is basically way to select one of the predetermined linker scripts of the GNU linker.

Linux user-space applications

Linux user-space applications are loaded by the dynamically linker in their own virtual memory address space, where they do not collide with other applications and it is a duty of dynamic linker to make sure that application doesn't collide with libraries it uses (if any). In most cases there is no need to use custom linker scripts.

Baremetal applications

Baremetal applications are loaded into target memory by debugger or by application bootloader or are already in the ROM mapped to specific location. If memory map used by linker is invalid that would mean that application will be loaded into the non-existing memory or will overwrite some another memory - depending on particular circumstances that would cause immediate failure on invalid write to non-existing memory, delayed failure when application will try to execute code from non-existing memory, or an unpredictable behaviour if application has overwritten something else.

Default linker emulation

Default linker emulation for ARC baremetal toolchain would put all loadable ELF sections as a consecutive region, starting with address 0x0. This is usually enough for an application prototyping, however real systems often has a more complex memory maps. Application linked with default linker emulation may not run on systems with CCMs and it is unlikely to run on systems with external memory if it is mapped to address other than 0x0. If system has some of it's memories mapped to 0x0 this memory may be overwritten by the debugger or application loader when it will be loading application into target - this may cause undesired effects. Default linker emulation also puts interrupt vector table (`.ivt` section) between code and data sections which is rarely reflects a reality and also default linker emulation doesn't align `.ivt` properly (address of interrupt vector table in ARC processors must be 1KiB-aligned). Therefore default linker emulation is not appropriate if application should handle interrupts. So default linker emulation can be used safely only with applications that don't handle interrupts and only on simulations that simulate whole address space, like following templates: em6_dmips, em6_gp, em6_mini, em7d_nrg, em7d_voice_audio, em11d_nrg, em11d_voice_audio, hs36_base, hs36, hs38_base, hs38, hs38_full, hs38_slc_full.

arcv2elfx linker emulation

For cases where default linker emulation is not enough there is an `arcv2elfx` linker emulation, which provides an ability to specify custom memory map to linker without the need to write a complete linker scripts. To use it pass option `-marcv2elfx` to the linker, but note that when invoking gcc driver it is required to specify this option as `-Wl,-marcv2elfx`, so that compiler driver would know that this is an option to pass to the linker, and not a machine-specific compiler option. When this option is present, linker will try to open a file named `memory.x`. Linker searches for this file in current working directory and in directories listed via `-L` option, but unfortunately there is no way to pass custom file name to the linker. `memory.x` must specify base addresses and sizes of memory regions where to put code and data sections. It also specifies parameters of heap and stack sections.

For example, here is a sample `memory.x` map for `hs34.tcf` template:

```
MEMORY {
    ICCM0    : ORIGIN = 0x00000000, LENGTH = 0x00004000
    DCCM     : ORIGIN = 0x80000000, LENGTH = 0x00004000
}

REGION_ALIAS("startup", ICCM0)
REGION_ALIAS("text", ICCM0)
REGION_ALIAS("data", DCCM)
REGION_ALIAS("sdata", DCCM)

PROVIDE (__stack_top = (0x80003fff & -4 ));
PROVIDE (__end_heap =  (0x80003fff ));
```

This `memory.x` consists of three logical sections. First sections `MEMORY` specifies a list of memory regions - their base address and size. Names of those regions can be arbitrary, and also it may describe regions that are not directly used by the linker. Second sections describes `REGION_ALIAS` es - this section translates arbitrary region names to standard region names expected by linker emulation. There are four such regions:

- `startup` for interrupt vector table and initialization code. Typically it should be mapped to the address 0x0. If interrupt vector is mapped to a different address, then in addition to respective value in `memory.x` it is required to pass an option `--defsym=ivtbase_addr=<your_ivt_address>` to the linker. If linker is invoked through the GCC driver, then the option should be prefixed with `-Wl,`.
- `text` is a region where code will be located.
- `data` is a regions where data will be located (unsurprisingly).
- `sdata` is a region where small data section will be located.

Finally two symbols are provided to specify end of data region in memory - `__stack_top` and `__end_heap`. They effectively point to same address, although `__stack_top` should be 4-byte aligned. `__stack_top` is a location where stack starts and it will grow downward. Heap starts at the address immediately following end of data sections (`.noinit` section to be exact) and grows upward to `__end_heap`. Therefore heap and stack grow towards each other and eventually may collide and overwrite each over. This linker emulation doesn't provide any protection against this scenario.

Custom linker scripts

In many cases neither default linker emulation, nor `arcv2elfx` are enough to describe memory map of a system, therefore it would be needed to write a custom linker script. Please consult GNU linker User manual for details. Default linker scripts can be found in `arc-elf32/lib/ldscripts` folder in toolchain installation directory.

Using TCF

General sescription

Currently GNU toolchain has a partial support for TCF, however it is not complete and in particular scenarios TCFs cannot be used as-is.

If you are using Eclipse IDE for ARC, please refer to a Building User Guide. Eclipse IDE for ARC supports only GCC compiler and GNU linker script sections of TCF, it doesn't support preprocessor defines sections as of version 2016.03.

If you are using GNU toolchain without IDE on Linux hosts you can use a special script **arc-elf32-tcf-gcc** (for big-endian toolchain this file has **arceb-** prefix) that is located in the same `bin` directory as rest of the toolchain executable files. This executable accepts all of the same options as GCC driver and also an option `--tcf <PATH/TO/TCF>`. **arc-elf32-tcf-gcc** will extract compiler options, linker script and preprocessor defines from TCF and will pass them to GCC along with other options.

- GCC options from `gcc_compiler` section will be passed as-is, but can be overridden by `-m<something>` options passed directly to **arc-elf32-tcf-gcc**.
- GNU linker script will be extracted from `gnu_linker_command_file` will be used as a memory.x file for `-Wl,marcv2elfx` linker emulation. Option `-Wl,-marcv2elfx` is added by this wrapper - there is no need to pass it explicitly.
- Preprocessor defines from section `C_defines` will be passed with `-include` option of GCC.

arc-elf32-tcf-gcc is a Perl script that requires `XML:LibXML` package. It is likely to work on most Linux hosts, however it will not work on Windows hosts, unless Perl with required library has been installed and added to the `PATH` environment variable. TCF is a text file in XML format, so in case of need it is trivial to extract compiler flags and linker script from TCF and use them directly with GCC and ld without IDE or wrapper script.

Value of `-mcpu=` option is selected by TCF generator to have best match with the target processor. This option Understanding GCC -mcpu option not only sets various hardware options but also selects a particular build of standard library. Values of hardware extensions can be overridden with individual `-m*` options, but that will not change standard library to a matching build - it still will use standard library build selected by `-mcpu=` value.

Compiler options

GCC options are stored in the `gcc_compiler` section of TCF. These options are passed to GCC as-is. These are "machine-specific" options applicable only to ARC, and which define configuration of target architecture - which of the optional hardware extensions (like bitscan instructions, barrel shifter instructions, etc) are present. Application that uses hardware extensions will not work on ARC processor without those extensions - there will be an Illegal instruction exception (although application may emulate instruction via handling of this exception, but that is out of scope of this document). Application that doesn't use hardware extensions present in the target ARC processor might be ineffective, if those extensions allow more optimal implementation of same algorithm. Usually hardware extensions allow improvement of both code size and performance at the expense of increased gate count, with all respective consequences.

When TCF is selected in the IDE respective compiler options are disabled in GUI and cannot be changed by user. However if TCF is deselected those options remain at selected values, so it is possible to "import" options from TCF and then modify it for particular need.

When using **arc-elf32-tcf-gcc** compiler options passed to this wrapper script has a higher precedence then options in TCF, so it is possible to use TCF as a "baseline" and then modify if needed.

Memory map

Please refer to main page about GNU linker for ARC Linker scripts and memory.x files for more details.

TCF doesn't contain a linker script for GNU linker in the strict meaning of this term. Instead TCF contains a special memory map, which can be used together with a linker emulation called **arcv2elfx**. This linker emulation reads a special file called `memory.x` to get several defines which denote location of particular memory areas, and then emulation allocates ELF sections to those areas. So, for example, `memory.x` may specify address and size of ICCM and DCCM memories and linker would put code sections into ICCM and data sections to DCCM. TCF contains this `memory.x` file as content of `gnu_linker_command_file` section. IDE and **arc-elf32-tcf-gcc** simply create this file and specify to linker to use **arcv2elfx** emulation. This is done by passing option `-marcv2elfx` to linker,

but note that when invoking gcc driver it is required to specify this option as `-Wl,-marcv2elfx`, so driver would know that this is an option to pass to linker.

It is very important that memory map in TCF matches the one in the hardware, otherwise application will not work. By default linker places all application code and data as a continuous sections starting from address 0x0. Designs with CCMs usually has ICCM mapped at address 0x0, and DCCM at addresses >= 0x8000_0000 (or simply an upper half of address space, which can be less then 32 bits wide). If application has both code and data put into ICCM, it may technically work (load/store unit in ARC has a port to ICCM), however this underutilizes DCCM and creates a risk of memory overflow where code and data will not fit into the ICCM, so overflown content will be lost, likely causing an error message in simulator or in debugger. For this reason it is recommended to use memory.x file from TCF when linking applications that use CCM memory. Typically TCF-generator would automatically assign instruction memory area to ICCM and data memory area to DCCM, because parameters of those memories can be read from BCRs, although it doesn't support such features as ICCM1 or NV ICCM.

When memory is connected via external memory bus TCF-generator cannot know where memory will be actually located, so it will put all sections continuously, starting from address 0. This is basically same as what happens when no memory map has been passed to linker. Therefore memory map in such TCF is effectively useless, instead it is needed to manually enter a proper memory map into "gnu_linker_command_file" section. However when using an nSIM simulator such TCF will work nice, as it will make nSIM simulate whole address space, so there is no risk that application will be loaded into unexisting address.

When using IDE there is an option to ignore memory map specified in TCF and use default memory mapping or custom linker script. This is the default setting - to ignore linker script embedded into TCF. However if target design uses closely-coupled memories then it is highly advised to use memory map (embedded into TCF or manually written).

C preprocessor defines

TCF section `C_defines` contains preprocessor defines that specify presence of various hardware optional extensions and values of Build Configuration Registers. **arc-elf32-tcf-gcc** wrapper extracts content of this section into temporary file and includes into compiled files via `-include` option of GCC.

arc-elf32-tcf-gcc options

--compiler
: Overwrites the default compiler name. The compiler tool chain needs to be in the PATH. Default value depends on the name of this file - it will call compiler that has the same name, only without -tcf part. Therefore:

  - arc-elf32-tcf-gcc -> arc-elf32-gcc
  - arceb-elf32-tcf-gcc -> arceb-elf32-gcc
  - arc-linux-tcf-gcc -> arc-linux-gcc
  - arceb-linux-tcf-gcc -> arceb-linux-gcc
  - arc-a-b-tcf-gcc -> arc-a-b-gcc
  - arc-tcf-elf32-tcf-gcc -> arc-tcf-elf32-gcc

--tcf
: The name and the location of the TCF file.

--verbose
: Verbose output. Prints the compiler invokation command.

Debugging baremetal applications

Using GNU Toolchain to Debug Applications on EM Starter Kit

Prerequisites

Toolchain for Linux and Windows hosts can be downloaded from the GNU Toolchain Releases page. For Linux hosts there is a choice between complete tarballs that include toolchain, IDE and OpenOCD (like installer for Windows), and tarballs that include toolchain only.

In order to use OpenOCD on Windows it is required to install appropriate WinUSB drivers, see How to Use OpenOCD on Windows for details.

Building an application

To learn how to build and debug application with Eclipse IDE, please use ARC GNU IDE manual.

Different core templates in EM Starter Kit use different memory maps, so different memory map files are required to compile applications that work properly on those configurations. This “toolchain” repository includes memory maps for all supported EM Starter Kit versions and configurations. They can be found at https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/tree/arc-staging/extras/dev_systems Memory map files in that directory have `.x` extension and file to be used should be renamed to `memory.x`, because `arcv2elfx` linker emulation doesn’t support ability to override that file name. Please refer to Linker scripts and memory.x files for more details about `memory.x` files.

For example for EM Starter Kit v2.3 EM7D to build an application:

```
$ cp -a toolchain/extras/dev_systems/sk2.3_em7d.x memory.x
$ arc-elf32-gcc -Wl,-marcv2elfx --specs=nosys.specs -mcpu=em4_dmips -O2 -g \
     test.c -o test.elf
```

List of compiler flags corresponding to particular CPUs

EM SK	CPU	Flags
v1	EM4	-mcpu=em4_dmips -mmpy-option=wlh5
	EM6	-mcpu=em4_dmips -mmpy-option=wlh5
v2.0	EM5D	-mcpu=em4 -mswap -mnorm -mmpy-option=wlh3 -mbarrel-shifter
	EM7D	-mcpu=em4 -mswap -mnorm -mmpy-option=wlh3 -mbarrel-shifter
	EM7DFPU	-mcpu=em4 -mswap -mnorm -mmpy-option=wlh3 -mbarrel-shifter -mfpu=fpuda_all
v2.1	EM5D	-mcpu=em4_dmips -mmpy-option=wlh3
	EM7D	-mcpu=em4_dmips -mmpy-option=wlh3
	EM7DFPU	-mcpu=em4_fpuda -mmpy-option=wlh3
v2.2	EM7D	-mcpu=em4_dmips
	EM9D	-mcpu=em4_fpus -mfpu=fpus_all
	EM11D	-mcpu=em4_fpuda -mfpu=fpuda_all
v2.3	EM7D	-mcpu=em4_dmips
	EM9D	-mcpu=em4_fpus -mfpu=fpus_all
	EM11D	-mcpu=em4_fpuda -mfpu=fpuda_all

C library for GNU Toolchain for ARC provides basic support for UART module of EM Starter Kit, which allows to use standard C function for input and output: `printf()`, `scanf()`, etc. Memory map files are also provided for processor configurations of EM Starter Kit. Both of those features are available via special “specs” files: `emsk_em9d.specs` and `emsk_em11d.specs` (there is no separate file for EM7D of EM Starter Kit, it is identical to EM9D). Usage example is following:

```
$ cat hello_world.c
#include <stdio.h>
int main() {
    printf("hello world\n");
    return 0;
}

$ arc-elf32-gcc --specs=emsk_em9d.specs -mcpu=em4_dmips hello_world.c
```

Note that it is still required to specify valid `-mcpu` option value - it is not set by the specs file.

Running an application with OpenOCD

Starting OpenOCD

Parameters of a particular target board are described in the OpenOCD configuration files. OpenOCD repository from Synopsys already includes several configuration files made specifically for Synopsys own development platforms: ARC EM Starter Kit and ARC SDP. Due to differences between different versions of ARC EM Starter Kit hardware, there are separate configuration files for different ARC EM Starter Kit versions:

- `snps_em_sk_v1.cfg` - for ARC EM Starter Kit v1.x.
- `snps_em_sk_v2.1.cfg` - for ARC EM Starter Kit versions 2.0 and 2.1.
- `snps_em_sk_v2.2.cfg` - for ARC EM Starter Kit version 2.2.
- `snps_em_sk_v2.3.cfg` - for ARC EM Starter Kit version 2.3.
- `snps_em_sk.cfg` - this is a configuration for ARC EM Starter Kit 2.0 and 2.1, preserved for compatibility.

Following documentation would assume the usage of the latest ARC EM Starter Kit version 2.3 which is similar to 2.2.

Start OpenOCD:

```
# On Linux (for manually built OpenOCD):
$ openocd  -c 'gdb_port 49101' -f board/snps_em_sk_v2.3.cfg

# On Linux (for prebuilt OpenOCD from IDE package):
$ $ide_dir/bin/openocd -s $ide_dir/share/openocd/scripts \
    -c 'gdb_port 49101' -f board/snps_em_sk_v2.3.cfg

@rem on Windows:
> openocd -s C:\arc_gnu\share\openocd\scripts -c "gdb_port 49101" ^
  -f board\snps_em_sk_v2.3.cfg
```

OpenOCD will be waiting for GDB connections on TCP port specified as an argument to `gdb_port` command, in this example it is 49101. When `gdb_port` command hasn’t been specified, OpenOCD will use its default port, which is 3333, however this port might be already occupied by some other software. In our experience we had a case, where port 3333 has been occupied, however no error messages has been printed but OpenOCD and GDB wasn’t printing anything useful as well, instead it was just printing some ambiguous error messages after timeout. In that case another application was occupying TCP port only on localhost address, thus OpenOCD was able to start listening on other IP addresses of system, and it was possible to connect GDB to it using that another IP address. Thus it is recommended to use TCP ports which are unlikely to be used by anything, like 49001-49150, which are not assigned to any application.

OpenOCD can be closed by CTRL+C. It is also possible to start OpenOCD from Eclipse as an external application.

Connecting GDB to OpenOCD

Write a sample application:

```
/* simple.c */
int main(void) {
    int a, b, c;
    a = 1;
    b = 2;
    c = a + b;
    return c;
}
```

Compile it - refer to “Building application” section for details, creation of `memory.x` is not shown in this example:

```
$ arc-elf32-gcc -Wl,-marcv2elfx --specs=nosys.specs -mcpu=em4_dmips -O2 -g \
    simple.c -o simple_sk2.3_em7d.elf
```

Start GDB, connect to target and run it:

```
$ arc-elf32-gdb --quiet simple_sk2.1_em5d.elf
# Connect. Replace 3333 with port of your choice if you changed it when starting OpenOCD
(gdb) target remote :3333
# Increase timeout, because OpenOCD sometimes can be slow
(gdb) set remotetimeout 15
# Load application into target
(gdb) load
# Go to start of main function
(gdb) tbreak main
(gdb) continue
# Resume with usual GDB commands
(gdb) step
(gdb) next
# Go to end of the application
(gdb) tbreak exit
(gdb) continue
# For example, check exit code of application
(gdb) info reg r0
```

Execution should stop at function `exit`. Value of register `r0` should be `3`.

Known issues and limitations

- Out of the box it is impossible to perform any input/output operations, like printf, scanf, file IO, etc.
    - When using an nSIM hostlink (GCC option `--specs=nsim.specs`), calling any of those function in application will result in a hang (unhandled system call to be exact).
    - When using libnosys (`--specs=nosys.specs`), standard IO functions will simply do nothing - they will set `errno = ENOSYS` and return -1 at most.
    - It is possible to use UART for text console I/O operations, but that is not implemented by default in GNU toolchain. Consult EM Starter Kit documentation and examples for details.
- Bare metal applications has nowhere to exit, and default implementation of exit is an infinite loop. To catch exit from application you should set breakpoint at function `exit` like in the example.

Using OpenOCD with AXS SDP

Synopsys DesignWare ARC Software Development Platforms (SDP) is a family of standalone platforms that enable software development, debugging and profiling. More information can be found at Synopsys web-site.

To debug applications on the AXS10x software development platforms you can use OpenOCD. Please consult with OpenOCD readme for instructions to download, build and install it.

AXS SDP consists of a mainboard and one of the CPU cards:

- AXS101 uses AXC001 CPU card;
- AXS102 uses AXC002 CPU card;
- AXS103 uses AXC003 CPU card.

For AXS103 currently two firmware releaseses are supported:

- Release 1.2: contains firmware for ARC HS36 CPU and ARC dualcore HS38 CPU
- Release 1.3: contains firmware for ARC HS47D CPU and ARC dualcore HS48 CPU

Prerequisites

Binary toolchain releases can be downloaded from the GNU Toolchain Releases page. For Linux hosts there is a choice between complete tarballs that include toolchain, IDE and OpenOCD, and tarballs that include toolchain only. For Windows hosts there is only a single installer, that contains all of the Toolchain components and allows to select which ones to install.

Note

In order to use OpenOCD it is required to install appropriate WinUSB drivers, see How to Use OpenOCD on Windows for details.

Building an application

To learn how to build and debug application with Eclipse IDE, please use ARC GNU IDE manual.

A memory map appropriate to the selected board should be used to link applications. This “toolchain” repository includes memory maps for all ARC SDP systems. They can be found in the tree. Memory map files in that directory have `.x` extension and file to be used should be renamed to `memory.x`, because `arcv2elfx` linker emulation doesn’t support ability to override that file name. Please refer to linker page Linker scripts and memory.x files for more details about `memory.x` files.

For example to build an application for AXS103/HS36:

```
$ cp -a toolchain/extras/dev_systems/axs103.x memory.x
$ arc-elf32-gcc -Wl,-marcv2elfx --specs=nosys.specs -O2 -g \
    -mcpu=hs38_linux test.c -o test.elf
```

List of compiler flags corresponding to particular CPUs

AXS	CPU	Flags
101	EM6	-mcpu=em4_dmips -mmpy-option=3
	ARC770	-mcpu=arc700
	AS221	-mcpu=arc600_mul32x16

AXS	CPU	Flags
102	HS34	-mcpu=hs -mdiv-rem -mmpy-option=9 -mll64 -mfpu=fpud_all
	HS36	-mcpu=hs -mdiv-rem -mmpy-option=9 -mll64 -mfpu=fpud_all
103	HS36	-mcpu=hs38_linux
	HS38	-mcpu=hs38_linux
	HS47	-mcpu=hs4x_rel31
	HS48	-mcpu=hs4x_rel31

Board configuration

First it is required to set jumpers and switches on the mainboard:

- PROG_M0 (JP1508), PROG_M1 (JP1401) and BS_EN (JP1507) jumpers should be removed (it is their default position);
- PROG_SRC (JP1403) jumper should be removed (default); DEBUG_SRC (JP1402) jumper should be placed (default); SW2501, SW2502, SW2503 and SW2401 should be set to their default positions according to the AXC00x CPU Card User Guides depending on what CPU card is being used. Following changes should be applied then:
    - SW2401.10 switch should be set to “1” (moved to the left), this will configure bootloader to setup clocks, memory maps and initialize DDR SDRAM and then halt a CPU.
    - For the core you are going to debug choose “Boot mode” type “Autonomously”, this is done by moving top two switches of the respective switch block to the position “1”. Alternatively, if you leave default boot mode “By CPU Start Button” you need to press “Start” button for this CPU, before trying to connect to it with the OpenOCD.

Configuration of the JTAG chain on the CPU card must match the configuration in the OpenOCD. By default OpenOCD is configured to expect complete JTAG chain that includes all of the CPU cores available on the card.

- For the AXC001 card jumpers TSEL0 and TSEL1 should be set.
- For the AXC002 card jumpers JP1200 and JP1201 should be set.
- For the AXC003 card it is not possible to modify JTAG chain directly.

Reset board configuration after changing jumpers or switch position, for this press “Board RST” button SW2410 near the power switch. Two seven-segment displays should show a number respective to the core that is selected to start autonomously. Dot should appear on the first display as well, to notify that bootloader was executed in bypass mode. To sum it up, for the AXS101 following numbers should appear:

- 1.0 for the AS221 core 1
- 2.0 for the AS221 core 2
- 3.0 for the EM6
- 4.0 for the ARC 770D.

For the AXS102 following numbers should appear:

- 1.0 for the HS34
- 2.0 for the HS36.

For the AXS103 firmware ver 1.2 following numbers should appear:

- 1.0 for the HS36
- 2.0 for the HS34

- 3.0 for the HS38 (core 0)
- 4.0 for the HS38 (core 1)

For the AXS103 firmware ver 1.3 following numbers should appear:

- 1.0 for the HS47D
- 3.0 for the HS48 (core 0)
- 4.0 for the HS48 (core 1)

Running OpenOCD

Run OpenOCD for the AXS101 platform:

```
$ openocd -f board/snps_axs101.cfg
```

Or run OpenOCD for the AXS102 platform:

```
$ openocd -f board/snps_axs102.cfg
```

AXS103 SDP supports different core configurations, so while in AXS101 and AXS102 there is a chain of several cores, which can operate independently, in AXS103 one of the particular configurations is chosen at startup and it is not possible to modify chain via jumpers. As a result, different OpenOCD configuration files should be used depending on whether AXS103 is configured to implement HS36 or to implement HS38.

To run OpenOCD for the AXS103 platform with HS36:

```
$ openocd -f board/snps_axs103_hs36.cfg
```

To run OpenOCD for the AXS103 platform with HS38x2:

```
$ openocd -f board/snps_axs103_hs38.cfg
```

To run OpenOCD for the AXS103 platform with HS47D:

```
$ openocd -f board/snps_axs103_hs47D.cfg
```

To run OpenOCD for the AXS103 platform with HS48x2:

```
$ openocd -f board/snps_axs103_hs48.cfg
```

OpenOCD will open a GDBserver connection for each CPU core on target (4 for AXS101, 2 for AXS102, 1 or 2 for AXS103). GDBserver for the first core listens on the TCP port 3333, second on port 3334 and so on. Note that OpenOCD discovers cores in the reverse order to core position in the JTAG chain. Therefore for AXS101 port assignment is following:

- 3333 - ARC 770D
- 3334 - ARC EM
- 3335 - AS221 core 2
- 3336 - AS221 core 1.

For AXS102 ports are:

- 3333 - ARC HS36

- 3334 - ARC HS34.

For AXS103 HS38x2 or HS48x2 ports are:

- 3333 - ARC HS38 or HS48 core 1
- 3334 - ARC HS38 or HS48 core 0.

For AXS103 HS47D ports are:

- 3333 - ARC HS47D

Running GDB

Run GDB:

```
$ arc-elf32-gdb ./application.to.debug
```

Connect to the target GDB server:

```
(gdb) target remote <gdbserver-host>:<port-number>
```

where `<gdbserver-host>` is a hostname/IP-address of the host that runs OpenOCD (can be omitted if it is a localhost), and `<port-number>` is a number of port of the core you want to debug (see previous section).

In most cases it is needed to load application into the target:

```
(gdb) load
```

After that application is ready for debugging.

To debug several cores on the AXC00x card simultaneously, it is needed to start additional GDBs and connect to the required TCP ports. Cores are controlled independently from each other.

Advanced topics

Using standalone Digilent HS debug cable

It is possible to use standalone Digilent HS1 or HS2 debug cable instead of the FT2232 chip embedded into the AXS10x mainboard. Follow AXS10x mainboard manual to learn how to connect Digilent cable to mainboard. In the nutshell:

- Connect cable to the DEBUG1 6-pin connector right under the CPU card. TMS pin is on the left (closer to the JP1501 and JP1502 jumpers), VDD pin is on the right, closer to the HDMI connector.
- Disconnect JP1402 jumper.

Then modify board configuration file used (`board/snps_axs101.cfg`, `board/snps_axs102.cfg`, etc): replace “source” of `snps_sdp.cfg` with “source” of `digilent-hs1.cfg` or `digilent-hs2.cfg` file, depending on what is being used.

Then restart OpenOCD.

Using OpenOCD with only one core in the JTAG chain

In AXS101 and AXS102 it is possible to reduce JTAG chain on the CPU card to a single core.

Change positions of TSEL0/TSEL1 (on AXC001) or JP1200/JP1201 (on AXC002) to reduce JTAG chain to a particular core. Follow AXC00x CPU Card User Guide for details.

Then modify OpenOCD command line to notify it that some core is not in the JTAG chain, for example:

```
$ openocd -c 'set HAS_HS34 0' -f board/snps_axs102.cfg
```

In this case OpenOCD is notified that HS34 is not in the JTAG chain of the AXC002 card. Important notes:

- Option `-c 'set HAS_XXX 0'` must precede option `-f`, because they are executed in the order they appear.
- By default all such variables are set `1`, so it is required to disable each core one-by-one. For example, for AXS101 it is required to set two variables. * Alternative solution is to modify `target/snps_axc001.cfg` or `target/snps_axc002.cfg` files to suit exact configuration, in this case there will be no need to set variables each time, when starting OpenOCD.

Those variables are used in the `target/snps_axc001.cfg` file: `HAS_EM6`, `HAS_770` and `HAS_AS221` (it is not possible to configure AXC001 to contain only single ARC 600 core in the JTAG chain). Those variables are used in the `target/snps_axc002.cfg` file: `HAS_HS34` and `HAS_HS36`.

When JTAG chain is modified, TCP port number for OpenOCD is modified accordingly. If only one core is in the chain, than it is assigned 3333 TCP port number. In case of AS221 TCP port 3333 is assigned to core 2, while port 3334 is assigned to core 1.

Troubleshooting

- **OpenOCD prints “JTAG scan chain interrogation failed: all ones”, then there is a lot of messages “Warn : target is still running!”.**

  An invalid JTAG adapter configuration is used: SDP USB data-port is used with configuration for standalone Digilent-HS cable, or vice versa. To resolve problem fix file `board/snps_axs10{1,2}.cfg` or `board/snps_axs103_hs36.cfg` depending on what board is being used.

- **OpenOCD prints “JTAG scan chain interrogation failed: all zeros”.**

  It is likely that position of JP1402 jumper does not match the debug interface you are trying to use. Remove jumper if you are using external debug cable, or place jumper if you are using embedded FT2232 chip.

- **OpenOCD prints that is has found “UNEXPECTED” device in the JTAG chain.**

  This means that OpenOCD configuration of JTAG chain does not match settings of jumpers on your CPU card.

- **I am loading application into target memory, however memory is still all zeros.**

  This might happen if you are using AXC001 CPU card and bootloader has not been executed. Either run bootloader for the selected core or configure core to start in autonomous mode and reset board after that - so bootloader will execute.

- **OpenOCD prints “target is still running!” after a CTRL+C has been done on the GDB client side.**

  There is an issue with EM6 core in AXS101 - after OpenOCD writes DEBUG.FH bit to do a force halt of the core, JTAG TAP of this core still occasionally returns a status that core is running, even though it has been halted. To avoid problem do not try to break execution with Ctrl+C when using EM6 on AXS101.

Using Ashling Opella-XD GDB server with AXS SDP

> **Note**
>
> The Ashling GDB Server software for ARC is implemented by Ashling and delivered as part of the Ashling Opella-XD probe for ARC processors product. This guide aims to provide all necessary information to successfully debug ARC applications using the GNU toolchain for ARC and the Ashling GDB server, however for all issues related to the Ashling GDB Server application, user should contact Ashling Microsystems Ltd. for further assistance.

Ashling GDB Server can be used to debug application running on the AXS10x family of software development platforms. It is recommended to use latest version of Ashling drivers and software package available.

Building an application

To learn how to build applications for AXS SDP, please refer to corresponding section of OpenOCD manual.

Board configuration

Board should be configured mostly the same way as for the OpenOCD, but it is required to change *JP1402* and *JP1403* jumpers - to debug with Opella-XD it is required to set *JP1403* and unset *JP1402*, while for OpenOCD it is otherwise. Refer to OpenOCD manual and to the User Guide of the AXC00x CPU card you are using for more details.

Running Ashling GDB Server

Note

Starting from Ashling ver. 1.2.6 **–device** option should contain specific cpu name of the board: “arc-600”, “arc-700”, “arc-em”, “arc-hs”. Using simple “arc” would cause an error.

Options of the Ashling GDB Server are described in its User Manual. It is highly recommended that users be familiar with Ashling GDB Server operation before proceeding. In a nutshell, to run GDB Server with multiple cores in the JTAG chain:

```
$ ./ash-arc-gdb-server --device arc --arc-reg-file <ARC_REG_FILE> \
    --scan-file arc2core.xml  --tap-number 1,2
```

Command for Ashling version starting from 1.2.6:

```
$ ./ash-arc-gdb-server --device arc-{CPU} --arc-reg-file <ARC_REG_FILE> \
    --scan-file arc2core.xml  --tap-number 1,2
```

where “arc-{CPU}” is equal to “arc-600”, “arc-700”, “arc-em”, “arc-hs”.

That will open GDB server connections on port 2331 (core 1) and 2332 (core 2). Use GDB to connect to the core you want to debug. `<ARC_REG_FILE>` is a path to a file with AUX register definitions for the core you are going to debug. Actual file that should be used depends on what target core is. A set of files can be found in this `toolchain` repository in `extras/opella-xd` directory. In this directory there are `arc600-cpu.xml`, `arc700-cpu.xml`, `arc-em-cpu.xml` and `arc-hs-cpu.xml` files for GDB server, direct link.

To run with AXS101 with all four cores in a chain

```
$ ./ash-arc-gdb-server --device arc --arc-reg-file <ARC_REG_FILE> \
    --scan-file arc4core.xml  --tap-number 1,2,3,4
```

Command for Ashling version starting from 1.2.6:

```
$ ./ash-arc-gdb-server --device arc-{CPU} --arc-reg-file <ARC_REG_FILE> \
    --scan-file arc4core.xml  --tap-number 1,2,3,4
```

where “arc-{CPU}” is equal to “arc-600”, “arc-700”, “arc-em”, “arc-hs”.

File `arc4core.xml` is not shipped with Ashling GDB Server, but can be easily created after looking at `arc2core.xml` and reading Ashling Opella-XD User Manual.

To run Ashling GDB Server with JTAG chain of a single core:

```
$ ./ash-arc-gdb-server --device arc --arc-reg-file <ARC_REG_FILE>
```

Command for Ashling version starting from 1.2.6:

```
$ ./ash-arc-gdb-server --device arc-{CPU} --arc-reg-file <ARC_REG_FILE>
```

where “arc-{CPU}” is equal to “arc-600”, “arc-700”, “arc-em”, “arc-hs”.

Option `--jtag-frequency ...MHz` can be passed to gdbserver to change JTAG frequency from default 1MHz. Rule of the thumb is that maximum frequency can be no bigger than half of the frequency, but for cores with external memory that value can be much lower. Most of the cores in different SDP models can work safely with JTAG frequencies around 10 ~ 12 MHz. ARC EM6 in the AXS101 is an exception - maximum recommended frequency is 5MHz.

Running GDB

Run GDB:

```
$ arc-elf32-gdb ./application.to.debug
```

Then it is required to specify description of target core that will be debugged with Ashling GDB Server.

Then it is required to specify XML target description file appropriate for the `ARC_REG_FILE` used to start Ashling GDB server. XML target description files for `arc600-cpu.xml`, `arc700-cpu.xml`, `arc-em-cpu.xml` and `arc-hs-cpu.xml` can be found in this `toolchain` repository in `extras/opella-xd`, direct link. Provided files are: `opella-arc600-tdesc.xml`, `opella-arc700-tdesc.xml`, `opella-arcem-tdesc.xml` and `opella-archs-tdesc.xml`. File `aux-minimal.xml` should be also downloaded from that folder and put into the same folder as `opella-*-tdesc.xml`. This file contains description common to all architectures and is included by all “tdesc” files. It is important that `ARC_REG_FILE` for Ashling GDB server and target description file for GDB match each other, so if Opella’s file has been modified, so should be the target description.:

```
(gdb) set tdesc filename <path/to/opella-CPU-tdesc.xml>
```

Connect to the target GDB server:

```
(gdb) target remote <gdbserver-host>:<port-number>
```

where `<gdbserver-host>` is a hostname/IP-address of the host that runs OpenOCD (can be omitted if it is localhost), and `<port-number>` is a number of port of the core you want to debug (see previous section).

In most cases you need to load application into the target:

```
(gdb) load
```

The system is now ready to debug the application.

To debug several cores on the AXC00x card simultaneously, start additional GDBs and connect to the required TCP ports. Cores are controlled independently from each other.

Known issues

- XML register file is specified only once in the GDB Server argument, that means that if your JTAG chain includes multiple cores of different model (e.g. ARC 700 and EM) you cannot debug them simultaneously, but you can debug multiple cores of they same type (e.g. all EM).
- GDB on Windows can't read XML files with Windows line endings (CR/LF) - tdesc XML file must be converted to UNIX line endings (LF).
- HS36 core of the AXS102 cannot be used when both cores are in the JTAG chain - if "resume" operation is initiated on the core, GDB Server and GDB will behave like it is running and never halting, but in reality it never started to run. To workaround this issue remove HS34 from the JTAG chain (remove JP1200 jumper on the AXC002 card, remove `--scan-file` and `--tap-number` options from Ashling GDB Server command line). If you need both HS34 and HS36 in the JTAG chain use OpenOCD instead of Ashling GDB Server. Why this problem happens is a mystery, since HS36 works without problems when it is single in the JTAG chain, and HS34 always work fine; this is likely a problem with Ashling GDB Server.
- In Opella software version of 1.0.6 prior to 1.0.6-D it has been observed that in some cases target core may hang on application load, if target has external memory attached. This happens when P-packet is disabled, and since P-packet should be disabled when using new GDB with those versions of Opella software, effectively it is not possible to use GDB >= 7.9 with Ashling GDBserver < 1.0.6-D to debug cores that employ external memory.
- In version of 1.0.6 it has been observed that breakpoint set at `main()` function of application may be not hit on first run in HS34 core in AXS102.
- In version 1.0.6-D it has been observed that gdbserver doesn't invalidate I$ of the second ARC 600 core of AXS101 - if this core hits a software breakpoint it gets stuck at it forever.

Known Issues of previous versions of Ashling software

- In version of Ashling software up to 1.0.5B, passing option `--tap-number 2` will cause GDB Server to print that it opened connection on port 2331 for core 2, however that is not true, instead GDB Server will create this connection for core 1. Therefore if your JTAG chain contains multiple ARC TAPs you _must_ specify all of them in the argument to `--tap-number` option.
- Up to version 1.0.5F there is an error in handling of 4-byte software breakpoints at 2-byte aligned addresses. For example in this sample of code attempt to set breakpoint at 0x2b2 will fail.:

```
0x000002b0 <+0>:    push_s     blink
0x000002b2 <+2>:    st.a       fp,[sp,-4]
0x000002b6 <+6>:    mov_s      fp,sp
0x000002b8 <+8>:    sub_s      sp,sp,16
```

- Big endian ARC v2 cores are not supported on versions prior to 1.0.5-F.

Linux applications

Debugging Linux Applications

This article describes how to debug user-space applications on the Linux on ARC.

Building toolchain

In most cases it should be enough to use binary distribution of GNU Toolchain for ARC, which can be downloaded from our Releases page. If toolchain in binary distribution doesn't fit some particular requirements, then instruction to build toolchain from source can be found in README.md file in the `toolchain` repository.

Building Linux

The simple guide to build kernel can be found in this manual page How to build Linux with GNU tools and run on simulator. More instructions can be found in ARC Linux wiki and in the Internet in general.

Configuring target system

Information in this section is not specific to ARC, it is given here just for convenience - there are other ways to achieve same result.

Configuring networking

> **Note**
>
> Ethernet model is not available in standalone nSIM simulation.

By default target system will not bring up networking device. To do this:

```
[arclinux] $ ifconfig eth0 up
```

If network to which board or virtual platform is attached has a DHCP server, then run DHCP client:

```
[arclinux] $ udhcpc
```

If there is no DHCP server, then configure networking manually:

```
[arclinux] $ ifconfig eth0 <IP_ADDRESS> netmask <IP_NETMASK>
[arclinux] $ route add default gw <NETWORK_GATEWAY> eth0
```

Where `<IP_ADDRESS>` is an IP address to assign to ARC Linux, `<IP_NETMASK>` is a mask of this network, `<NETWORK_GATEWAY>` is default gateway of network.

To gain access to the Internet DNS must servers must be configured. This is usually not required when using DHCP, because in this case information about DNS servers is provided via DHCP. To configure DNS manually, create `/etc/resolv.conf` which lists DNS servers by IP. For example:

```
nameserver 8.8.8.8
nameserver 8.8.4.4
```

That will connect ARC Linux to the network.

Configuring NFS

To ease process of delivering target application into the ARC Linux it is recommended to configure NFS share and mount it on the ARC Linux.

Install NFS server on the development host (in this example it is Ubuntu):

```
$ sudo apt-get install nfs-kernel-server
```

Edit `/etc/exports`: describe you public folder there. For example:

```
/home/arc/snps/pub *(rw,no_subtree_check,anonuid=1000,anongid=1000,all_squash)
```

Restart NFS server:

```
$ sudo service nfs-kernel-server restart
```

Open required ports in firewall. To make things easier this example will open *all* ports for the hosts in the `tap0` network:

```
$ sudo ufw allow from 192.168.218.0/24 to 192.168.218.1
```

Now you can mount your share on the target:

```
[arclinux] # mount -t nfs -o nolock,rw 192.168.218.1:/home/arc/snps/pub /mnt
```

Public share will be mounted to the `/mnt` directory.

Additional services

Another thing that might be useful is to have network services like telnet, ftp, etc, that will run on ARC Linux. First make sure that desired service is available in the Busybox configuration. Run `make menuconfig` from Busybox directory or `make busybox-menuconfig` if you are using Buildroot. Make sure that “inetd” server is enabled. Select required packages (telnet, ftpd, etc) and save configuration. Rebuild busybox (run `make busybox-rebuild` if you are using Buildroot).

Then configure inetd daemon. Refer to inetd documentation to learn how to do this. In the simple case it is required to create `/etc/inetd.conf` file on the target system with following contents:

```
ftp     stream  tcp nowait  root    /usr/sbin/ftpd      ftpd -w /
telnet  stream  tcp nowait  root    /usr/sbin/telnetd   telnetd -i -l /bin/sh
```

Thus inetd will allow connections to ftpd and telnetd servers on the target system. Other services can be added if required.

Rebuild and update rootfs and vmlinux. Start rebuilt system and run *inetd* to start inetd daemon on target:

```
[arclinux] $ inetd
```

Debugging applications with gdbserver

It is assumed that one or another way application to debug is on to the target system. Run application on target with gdbserver:

```
[arclinux] $ gdbserver :49101 <application-to-debug> [application arguments]
```

TCP port number could any port not occupied by another application. Then run GDB on the host:

```
$ arc-linux-gdb <application-to-debug>
```

Then set sysroot directory path. Sysroot is a “mirror” of the target system file system: it contains copies of the applications and shared libraries installed on the target system. Path to the sysroot directory should be set to allow GDB to step into shared libraries functions. Note that shared libraries and applications on the target system can be stripped from the debug symbols to preserve disk space, while files in the sysroot shouldn’t be stripped. In case of Buildroot-generated rootfs sysroot directory can be found under `<BUILDROOT_OUTPUT>/staging`.:

```
(gdb) set sysroot <SYSROOT_PATH>
```

Then connect to the remote gdbserver:

```
(gdb) target remote <TARGET_IP>:49101
```

You can find `<TARGET_IP>` via running `ifconfig` on the target system. TCP port must much the one used when starting up gdbserver. It is important that sysroot should be set before connecting to remote target, otherwise GDB might have issues with stepping into shared libraries functions.

Then you can your debug session as usual. In the simplest case:

```
(gdb) continue
```

Debugging applications with native GDB

Starting from GNU Toolchain for ARC release 2014.08 it is possible to build full GDB to run natively on ARC Linux. Starting from GNU Tooolchain for ARC release 2015.06 native GDB is automatically built for uClibc toolchain (can be disabled by `--no-native-gdb` option). In GNU Toolchain prebuilt tarballs native GDB binary can be found in sysroot directory: `arc-snps-linux-uclibc/sysroot/usr/bin/gdb`

With native GDB it is possible to debug applications the same way as it is done on the host system without gdbserver.

How to build Linux with GNU tools and run on simulator

This document describes how to build Linux kernel image from the perspective of toolchain developer. This document doesn't aim to replace more complete and thorough Linux-focused user guides and how to so. This document answers the single question "How to confirm, that I can build Linux kernel with *that* toolchain?"

To learn how to configure Linux, debug kernel itself and build extra software please see https://github.com/foss-for-synopsys-dwc-arc-processors/linux/wiki.

Prerequisites

- Host OS:
  - RHEL 6 or later
  - Ubuntu 14.04 LTS or later
- GNU tool chain for ARC:
  - 2014.12 or later for Linux for ARC HS
  - 4.8 or later for Linux for ARC 700
- make version at least 3.81
- rsync version at least 3.0
- git

Prerequisite packages can be installed on Ubuntu 14.04 with the following command:

```
# apt-get install texinfo byacc flex build-essential git
```

On RHEL 6/7 those can be installed with following command:

```
# yum groupinstall "Development Tools"
# yum install texinfo-tex byacc flex git
```

Overview

There are two essential components to get a working Linux kernel image: root file system and Linux image itself. For the sake of simplicity this guide assumes that root file system is embedded into the Linux image.

To generate root file system this guide will use Buildroot project, that automates this sort of things. Buildroot is capable to build Linux image itself, feature that is also used in this guide. Buildroot is also capable of building toolchain from the source, however this feature is not used in this guide, instead binary toolchain distributed by Synopsys will be used.

Configuring

Check Buildroot downloads page for latest release. This guide further assumes latest snapshot. Get Buildroot sources:

```
$ mkdir arc-2020.09-linux-guide
$ cd arc-2020.09-linux-guide
$ wget https://buildroot.org/downloads/buildroot-2020.08.1.tar.bz2
$ tar xf buildroot-2020.08.1.tar.bz2
```

To build Linux and rootfs Buildroot should be configured. For the purpose of this guide, a custom “defconfig” file will be created and then will be used to configure Buildroot. Custom “defconfig” file can be located anywhere and have any name. For example it can be `arc-2020.09-linux-guide/hs_defconfig`. Contents of this file should be following:

```
BR2_arcle=y
BR2_archs38=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM=y
BR2_TOOLCHAIN_EXTERNAL_DOWNLOAD=y
BR2_TOOLCHAIN_EXTERNAL_URL="https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/rel
BR2_TOOLCHAIN_EXTERNAL_GCC_10=y
BR2_TOOLCHAIN_EXTERNAL_HEADERS_5_7=y
BR2_TOOLCHAIN_EXTERNAL_LOCALE=y
BR2_TOOLCHAIN_EXTERNAL_HAS_SSP=y
BR2_TOOLCHAIN_EXTERNAL_CXX=y
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_DEFCONFIG="haps_hs"
BR2_LINUX_KERNEL_VMLINUX=y
BR2_PACKAGE_GDB=y
BR2_PACKAGE_GDB_DEBUGGER=y
BR2_TARGET_ROOTFS_INITRAMFS=y
# BR2_TARGET_ROOTFS_TAR is not set
```

Important notes about modifying Buildroot defconfig:

- `BR2_TOOLCHAIN_EXTERNAL_URL` should point to a valid URL of GNU Toolchain for ARC distributable.
- `BR2_TOOLCHAIN_EXTERNAL_HEADERS_X_XX` should be aligned to Linux headers version used for the toolchain build.

Toolchain version	Linux headers version
2020.09	5.7
2020.03	4.15
2019.09	4.15
2019.03	4.15
2018.09	4.15
2018.03	4.15

Toolchain version	Linux headers version
2017.09	4.12
2017.03	4.9
2016.09	4.8
2016.03	4.6
2015.06, 2015.12	3.18
earlier	3.13

This parameter identifies version of Linux that was used to build toolchain and is not related to version of Linux that will be *built by* the toolchain or where applications compiled by this toolchain will run.

- For building big endian linux you have to replace `BR2_arcle=y` to `BR2_arceb=y` and change value of `BR2_TOOLCHAIN_EXTERNAL_URL` to respective URL for your processor.
- Other Linux kernel defconfigs can be used.
- Building GDB or GDBserver is not necessary.

Building

To build Linux kernel image using that defconfig:

```
$ mkdir output_hs
$ cd buildroot-2020.08.01
$ make O=`readlink -e ../output_hs` defconfig DEFCONFIG=`readlink -e ../hs_defconfig`
$ cd ../output_hs
$ make
```

It’s necessary to pass an absolute path to Buildroot, because there is the issue with a relative path.

After that there will be Linux kernel image file `arc-2020.09-linux-guide/output/images/vmlinux`.

Running on nSIM

Linux configuration in the provided Buildroot defconfig is for the standalone nSIM. This kernel image can be run directly on nSIM, without any other additional software. Assuming current directory is `arc-2020.09-linux-guide`:

```
$ $NSIM_HOME/bin/nsimdrv -prop=nsim_isa_family=av2hs -prop=nsim_isa_core=3 -prop=chipid=0xffff -p
```

Username is `root` without a password. To halt target system issue `halt` command.

Add `-prop=nsim_fast=1` to props file if you have nSIM Pro license.

For more information visit this page: How to run ARC Linux kernel and debug

Using different Linux configuration

It is possible to change Linux configuration used via altering `BR2_LINUX_KERNEL_DEFCONFIG` property of Buildroot defconfig. For example to build kernel image for AXS103 SDP change its value to `axs103`. After that repeat steps from Building section of this document. Refer to ARC Linux documentation for more details about how to enable networking, HDMI and other hardware features of AXS10x SDP.

Notable defconfigs available for ARC: `axs101`, `axs103`, `axs103_smp`, `vdk_hs38_smp`, `hsdk`.

Using glibc toolchain

Configuration for glibc toolchain is fairly similar for uClibc, with only minor differences:

```
BR2_arcle=y
BR2_archs38=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM=y
BR2_TOOLCHAIN_EXTERNAL_DOWNLOAD=y
BR2_TOOLCHAIN_EXTERNAL_URL="https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/rel
BR2_TOOLCHAIN_EXTERNAL_GCC_10=y
BR2_TOOLCHAIN_EXTERNAL_HEADERS_5_7=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM_GLIBC=y
BR2_TOOLCHAIN_EXTERNAL_CXX=y
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_DEFCONFIG="haps_hs"
BR2_LINUX_KERNEL_VMLINUX=y
BR2_PACKAGE_GDB=y
BR2_PACKAGE_GDB_DEBUGGER=y
BR2_TARGET_ROOTFS_INITRAMFS=y
# BR2_TARGET_ROOTFS_TAR is not set
```

Linux for ARC 770 processors

Process of building kernel for ARC 770 is similar to what is for ARC HS. It is required only to change several option in Buildroot defconfig:

- `BR2_archs38=y` with `BR2_arc770d=y`
- `BR2_TOOLCHAIN_EXTERNAL_URL="https://github.com/foss-for-synopsys-dwc-arc-processor` with `BR2_TOOLCHAIN_EXTERNAL_URL="https://github.com/foss-for-synopsys-dwc-arc-processor`
- `BR2_LINUX_KERNEL_DEFCONFIG="haps_hs"` with `BR2_LINUX_KERNEL_DEFCONFIG="nsim_700"`

Then repeat steps from :ref`linux-building-label` section of this document to build Linux kernel image. To run this image in nSIM use next command:

```
$ $NSIM_HOME/bin/nsimdrv -prop=nsim_isa_family=a700 -prop=nsim_isa_atomic_option=1 -prop=nsim_mmu
```

Linux for ARC HS VDK

This section is specific to ARC HS VDK which is distributed along with nSIM (nSIM Pro license is required).

Buildroot defconfig for VDK differs from the one for a simple nSIM:

- Linux defconfig is `vdk_hs38_smp`.
- Ext2 file of root file system should be created, instead of being linked into the kernel.

With those changes Buildroot defconfig for ARC HS VDK is:

```
BR2_arcle=y
BR2_archs38=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM=y
BR2_TOOLCHAIN_EXTERNAL_DOWNLOAD=y
BR2_TOOLCHAIN_EXTERNAL_URL="https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/rel
BR2_TOOLCHAIN_EXTERNAL_GCC_10=y
BR2_TOOLCHAIN_EXTERNAL_HEADERS_5_7=y
BR2_TOOLCHAIN_EXTERNAL_LOCALE=y
BR2_TOOLCHAIN_EXTERNAL_HAS_SSP=y
BR2_TOOLCHAIN_EXTERNAL_CXX=y
BR2_LINUX_KERNEL=y
```

```
BR2_LINUX_KERNEL_DEFCONFIG="vdk_hs38_smp"
BR2_LINUX_KERNEL_VMLINUX=y
BR2_PACKAGE_GDB=y
BR2_PACKAGE_GDB_DEBUGGER=y
BR2_TARGET_ROOTFS_EXT2=y
# BR2_TARGET_ROOTFS_TAR is not set
```

Save this defconfig to some file (for example `vdk_defconfig`). Then use same process as in Building section.:

```
$ mkdir output_vdk
$ cd buildroot
$ make O=`readlink -e ../output_vdk` defconfig DEFCONFIG=<path-to-VDK-defconfig-file>
$ cd ../output_vdk
$ make
```

ARC HS VDK already includes Linux kernel image and root file system image. To replace them with your newly generated files:

```
$ cd <VDK-directory>/skins/ARC-Linux
$ mv rootfs.ARCv2.ext2{,.orig}
$ ln -s <path-to-Buildroot-output/images/rootfs.ext2 rootfs.ARCv2.ext2
$ mv ARCv2/vmlinux_smp{,.orig}
$ ln -s <path-to-Buildroot-output/images/vmlinux ARCv2/vmlinux_smp
```

Before running VDK if you wish to have a working networking connection on Linux for ARC system it is required to configure VDK VHub application. By default this application will pass all Ethernet packets to the VDK Ethernet model, however on busy networks that can be too much to handle in a model, therefore it is highly recommended to configure destination address filtering. Modify `VirtualAndRealWorldIO/VHub/vhub.conf`: : set `DestMACFilterEnable` to `true`, and append some random valid MAC address to the list of `DestMACFilter`, or use one of the MAC address examples in the list. This guide will use D8:D3:85:CF:D5:CE - this address is already in the list. Note that is has been observed that it is not possible to assign some addresses to Ethernet device model in VDK, instead of success there is an error “Cannot assign requested address”.

Note, that due to the way how VHub application works, it is impossible to connect to the Ethernet model from the host on which it runs on and vice versa. Therefore to use networking in target it is required to either have another host and communicate with it.

Run VHub application as root:

```
# VirtualAndRealWorldIO/VHub/vhub -f VirtualAndRealWorldIO/VHub/vhub.conf
```

In another console launch VDK:

```
$ . setup.sh
$ ./skins/ARC-Linux/start_interactive.tcl
```

After VDK will load, start simulation. After Linux kernel will boot, login into system via UART console: login `root`, no password. By default networking is switched off. Enable `eth0` device, configure it is use MAC from address configured in VHub:

```
[arclinux] # ifconfig eth0 hw ether d8:d3:85:cf:d5:ce
[arclinux] # ifconfig eth0 up
```

Linux kernel will emit errors about failed PTP initialization - those are expected. Assign IP address to the target system. This example uses DHCP:

```
[arclinux] # udhcpc eth0
```

Now it is possible to mount some NFS share and run applications from it:

```
[arclinux] # mount -t nfs public-nfs:/home/arc_user/pub /mnt
[arclinux] # /mnt/hello_world
```

Linux for AXS103 SDP

Build process using Buildroot is the same as for standalone nSIM. Buildroot defconfig is:

```
BR2_arcle=y
BR2_archs38=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM=y
BR2_TOOLCHAIN_EXTERNAL_DOWNLOAD=y
BR2_TOOLCHAIN_EXTERNAL_URL="https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/rel
BR2_TOOLCHAIN_EXTERNAL_GCC_10=y
BR2_TOOLCHAIN_EXTERNAL_HEADERS_5_7=y
BR2_TOOLCHAIN_EXTERNAL_LOCALE=y
BR2_TOOLCHAIN_EXTERNAL_HAS_SSP=y
BR2_TOOLCHAIN_EXTERNAL_CXX=y
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_DEFCONFIG="axs103_smp"
BR2_PACKAGE_GDB=y
BR2_PACKAGE_GDB_DEBUGGER=y
BR2_TARGET_ROOTFS_INITRAMFS=y
# BR2_TARGET_ROOTFS_TAR is not set
```

This defconfig will create a uImage file instead of vmlinux. Please refer to ARC Linux wiki for more details on using u-boot with AXS103.

Using KGDB to debug Linux

While user-space programs can be debugged with regular GDB (in combination with gdbserver), this is not the case for debugging the kernel. gdbserver is a user-space program itself and cannot control the kernel. KGDB, Kernel GDB, solves this by acting as a gdbserver that is inside the kernel.

Configuring the Kernel for KGDB

Your kernel configuration needs to have the following options set:

```
CONFIG_KGDB
CONFIG_KGDB_SERIAL_CONSOLE
```

Kernel command line

Use the kgdboc option on the kernel boot args to tell KGDB which serial port to use. Kernel bootargs can be modified in the DTS file or can be passed via bootloader if it is used.

Examples:

- One serial port, KGDB is shared with console: `console=ttyS0,115200n8 kgdboc=ttyS0,115200`
- Two serial ports, one for console, another for KGDB: `console=ttyS0,115200n8 kgdboc=ttyS1,115200`

These examples assume you want to attach gdb to the kernel at a later stage. Alternatively, you can add the kgdbwait option to the command line. With kgdbwait, the kernel waits for a debugger to attach at boot time. In the case of two serial ports, the kernel command line looks like the following:

```
console=ttyS0,115200n8 kgdboc=ttyS1,115200 kgdbwait
```

Connect from GDB

After the kernel is set up, you can start the debugging session. To connect to your target using a serial connection, you need to have a development PC with UART that runs GDB and a terminal program.

Stop the Kernel

First, stop the kernel on the target using a SysRq trigger. To do so, send a `remote break` command using your terminal program, followed by the character `g`: * using minicom: `Ctrl-a, f, g` * using Tera Term: `Alt-b, g`

You must also stop the kernel if you have two UARTs, even though one of the two UARTs is dedicated to KGDB.

Connect GDB

After stopping the kernel, connect GDB:

```
$ arc-elf32-gdb vmlinux
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
```

You are then connected to the target and can use GDB like any other program. For instance, you can set a breakpoint now using `b <identifier>` and then continue kernel execution again using `c`.

Other

How to run DejaGNU tests for ARC toolchain

This article describes how to run testsuites of GNU projects for ARC. This article covers only baremetal toolchain. Tests can be run on different platforms - simulators or hardware platforms.

Prerequisites

1. Get sources of the project you want to test.
2. Get source of the “toolchain” repository:

   ```
   $ git clone git@github.com:foss-for-synopsys-dwc-arc-processors/toolchain.git
   ```

3. Download free nSIM from webpage: https://www.synopsys.com/cgi-bin/dwarcnsim/req1.cgi. Note that requests are manually approved, hence it make take up to 2 workdays for download to be ready.
4. Build toolchain.

Preparing

Create a directory where artifacts and temporary files will be created:

```
$ mkdir tests
$ cd tests
```

Create a `site.exp` file in this directory. `toolchain/dejagnu/example_site.exp` can be used as an example. Note that this file is not intended to be used as-is, it must be edited for a particular case: `srcdir` and `arc_exec_prefix` variables must be set to location of toolchain sources and location of installed toolchain respectively. Variable `toolchain_sysroot_dir` in that file shouldn’t be set for baremetal toolchain testing.

Required environment variables are:

- `ARC_MULTILIB_OPTIONS` - should be set to compiler that to be tested. Note that first `-m<something>` should be omitted, but only first. So for example this variables might take value: `cpu=arcem -mnorm`.
- `DEJAGNU` - should be set to path to the `site.exp` file in "toolchain" repository.
- `PATH` - add toolchain installation location `bin` directory to PATH - some of the internal checks in the GNU testsuite often ignore tool variables set in site.exp and instead expect that tools are in the PATH.
- `NSIM_HOME` - should point to location where nSIM has been untarred.
- `ARC_GCC_COMPAT_SUITE` - set to 0 if you are not going to run compatibility tests.

Some actions are specific to particular GNU projects:

- Newlib requires:

```
$ mkdir targ-include
$ ln -s /home/user/arc_gnu/INSTALL/arc-elf32/include/newlib.h targ-include/
```

- GCC might require (for some tests):

```
$ pushd home/user/arc_gnu/gcc/
$ ./contrib/gcc_update --touch
$ popd
```

- libstdc++ requires:

```
$ export CXXFLAGS="-O2 -g"
```

- GDB requires:

```
$ testsuite=$/home/user/arc_gnu/gdb/testsuite
$ mkdir $(ls -1d $testsuite/gdb.* | grep -Po '(?<=\/)[^\/]+$')
```

Also arc-nsim.exp board will require an environment variable `ARC_NSIM_PROPS` to be set and to contain path to nSIM properties file that specifies ISA options.

Toolchain repository an example `run.sh` file that does some of those actions: `toolchain/dejagnu/example_run.sh`. Note that example file is not intended to run as-is - it should be modified for particular purpose.

Baremetal boards that run tests in the development systems, like arc-openocd.exp require a valid `memory.x` file in the current working directory.

Running

Now, that all is set, test suite can be run with:

```
$ runtest --tool=<project-to-test> --target_board=<board> \
  --target=arc-default-elf32
```

Where `<project-to-test>` can be: gcc, g++, binutils, ld, gas, newlib, libstdc++ or gdb. `board` for free nSIM can be arc-nsim.exp or arc-sim-nsimdrv.exp. The former runs nSIM as a GDBserver, while the latter will run nSIM as a standalone simulator, which is faster and more stable, but not suitable to run GDB testsuite.

If `example_run.sh` is being used, then assuming that it has been configured properly, then running is as simple as invoking it.

Compatibility tests

GCC contains a set of compatibility tests named `compat.exp`. It allows to test compatibility of ARC GNU gcc compiler and proprietary Synopsys MetaWare ccac compiler for ARC EM and ARC HS targets. If you want to run these tests it is necessary to configure additional variables in `site.exp` file:

- `set is_gcc_compat_suite "1"` - enable support of compatibility tests from gcc.
- `set ALT_CC_UNDER_TEST "path/to/ccac"`
- `set ALT_CXX_UNDER_TEST "path/to/ccac"`
- `set COMPAT_OPTIONS [list [list "options for gcc" "options for ccac"]]`
- `set COMPAT_SKIPS [list {ATTRIBUTE}]` - disable tests with packed structures to avoid unaligned access errors.

Then `runtest` program must be invoked with an additional option `compat.exp`:

```
$ runtest --tool=<project-to-test> --target_board=<board> \
  --target=arc-default-elf32 compat.exp
```

Also you can use `example_run.sh` and `example_site.exp` to simplify configuration and set these environment variables in `example_run.sh`:

- `runtestflags` - set to `compat.exp` to run compatibility tests only.
- `ARC_GCC_COMPAT_SUITE` - set to 1.
- `GCC_COMPAT_CCAC_PATH` - path to Synopsys MetaWare ccac executable.
- `GCC_COMPAT_GCC_OPTIONS` - options for gcc.
- `GCC_COMPAT_CCAC_OPTIONS` - options for ccac.

ARC-specific board options

Following options are supported by ARC DejaGNU scripts and are usually set in board files, for example in `dejagnu/baseboard/arc-sim-nsimdrv.exp`.

`arc,gdbserver_prog`
: Path to GDB server to use with `arc-nsim.exp`.

`arc,gdbserver_args`
: Argument to pass to GDB server used in `arc-nsim.exp`.

`arc,hostlink`
: Hostlink type to use. Can be `nsim` or empty/not set.

`arc,is_gcc_compat_suite`
: Whether this is a GCC *compat* testsuite or not. Boolean value.

`arc,openocd_prog`
: Path to OpenOCD application binary.

`arc,openocf_cfg`
: OpenOCD configuration file. Passed to `openocd` via option `-s` as-is.

`arc,openocd_log`
: Path to logfile for OpenoCD.

`arc,openocd_log_level`
: Level of OpenOCD verbosity. Integer from 0 to 3 inclusive.

ARC GNU IDE

The ARC GNU Eclipse IDE consists of the Eclipse IDE combined with an Eclipse CDT Managed Build Extension plug-in for the ARC GNU Toolchain and GDB embedded debugger plug-in for ARC, based on the Zylin Embedded CDT plug-in. The ARC GNU IDE supports the development of managed C/C++ applications for ARC processors using the ARC GNU toolchain for bare metal applications (elf32).

The ARC GNU IDE provides support for the following functionality:

- Support for the ARC EM, ARC HS, ARC 600 and ARC 700 Processors
- Support for little and big endian configurations
- Ability to create C/C++ projects using the ARC elf32 cross-compilation toolchain
- Configuration of toolchain parameters per project
- Configuration of individual options (such as preprocessor, optimization, warnings, libraries, and debugging levels) for each toolchain component:
    - GCC Compiler
    - GDB Debugger
    - GAS assembler
    - Size binutils utility, etc.
- Support for Synopsys EM Starter Kit and AXS10x.
- Configuration of debug and run configurations for supported FPGA Development Systems and debug probes (Digilent HS1/HS2 or Ashling Opella-XD).
- GDB-based debugging using **Debug** perspective providing detailed debug information (including breakpoints, variables, registers, and disassembly)

ARC GNU plugins for Eclipse have following requirements to the system:

- OS: Windows 10, Ubuntu Linux 16.04 LTS and CentOS 7 development host systems
- Eclipse 2018-12 (part of Windows installer)
- CDT version 9.6.0 (part of Windows installer)
- Java VM version >= 1.8 is required (part of Windows installer)

Note

Before you begin, refer to the EM Starter Kit guide and follow the instructions on how to connect EM Starter Kit to your PC. This is required for the Eclipse IDE GDB debugger to successfully download and debug programs on the target.

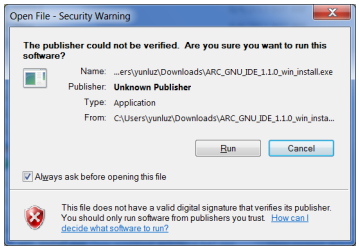
Getting Started

Installation

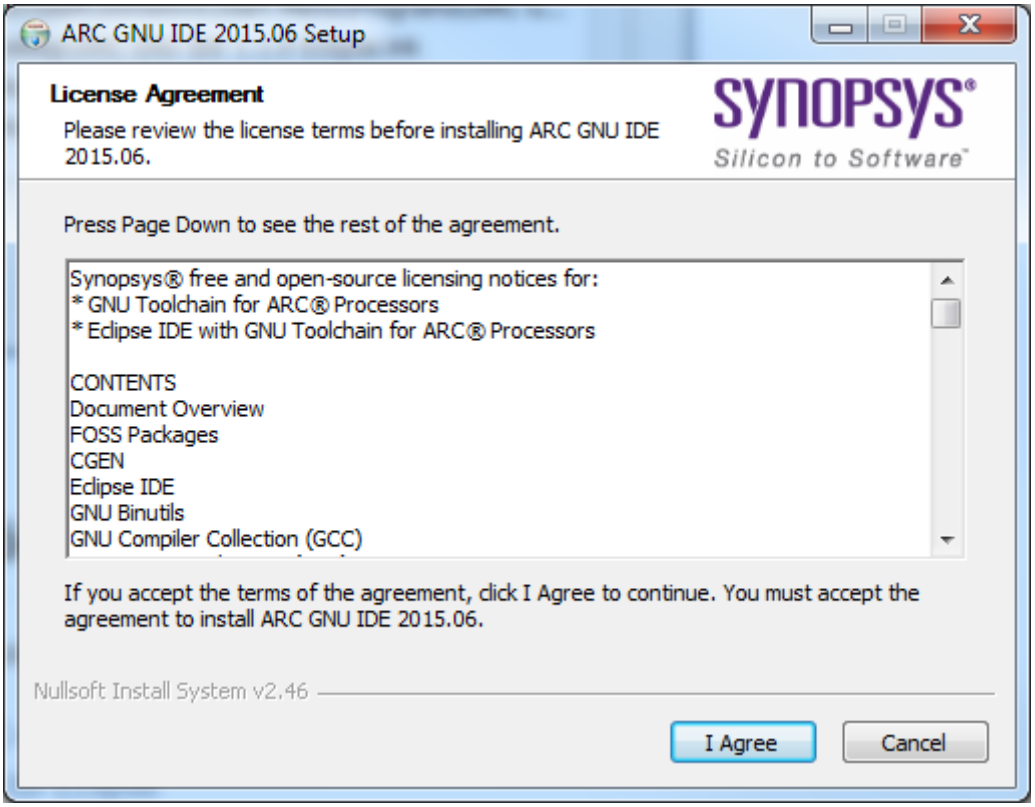
Using installer for Windows

Windows users are advised to use our Windows installer for Eclipse for GNU Toolchain for IDE, that can be downloaded from this releases page. Installer already contains all of the necessary components.

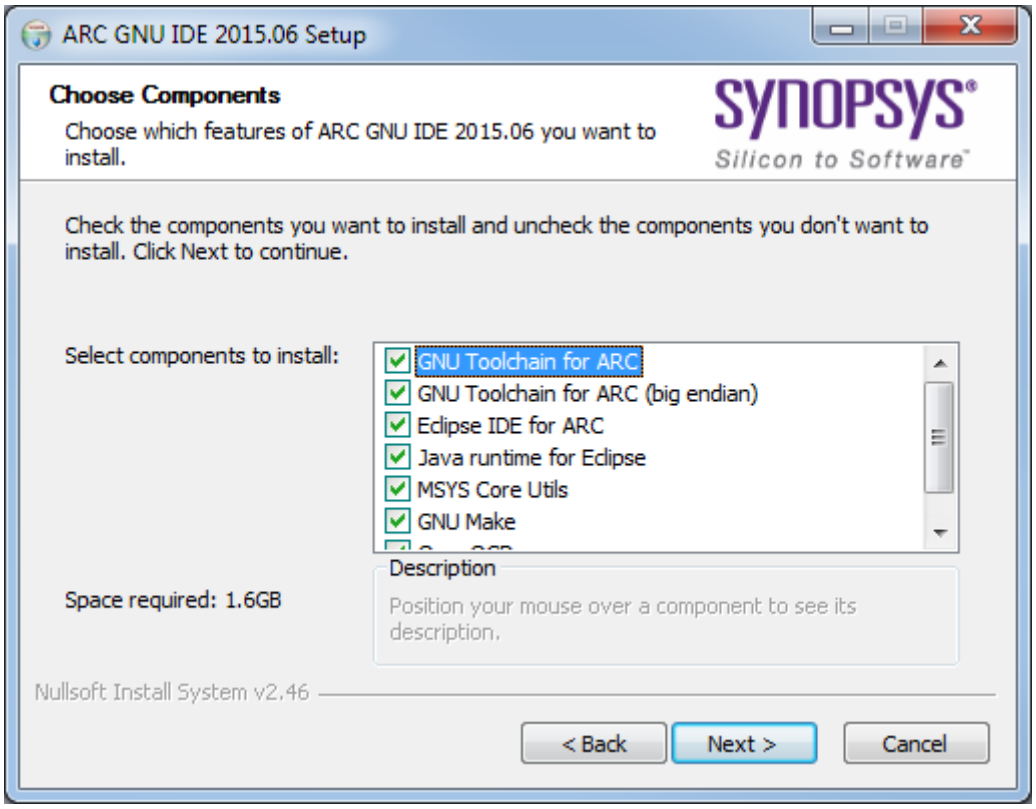
ARC GNU IDE should be installed in the path no longer than 50 characters and cannot contain white spaces.



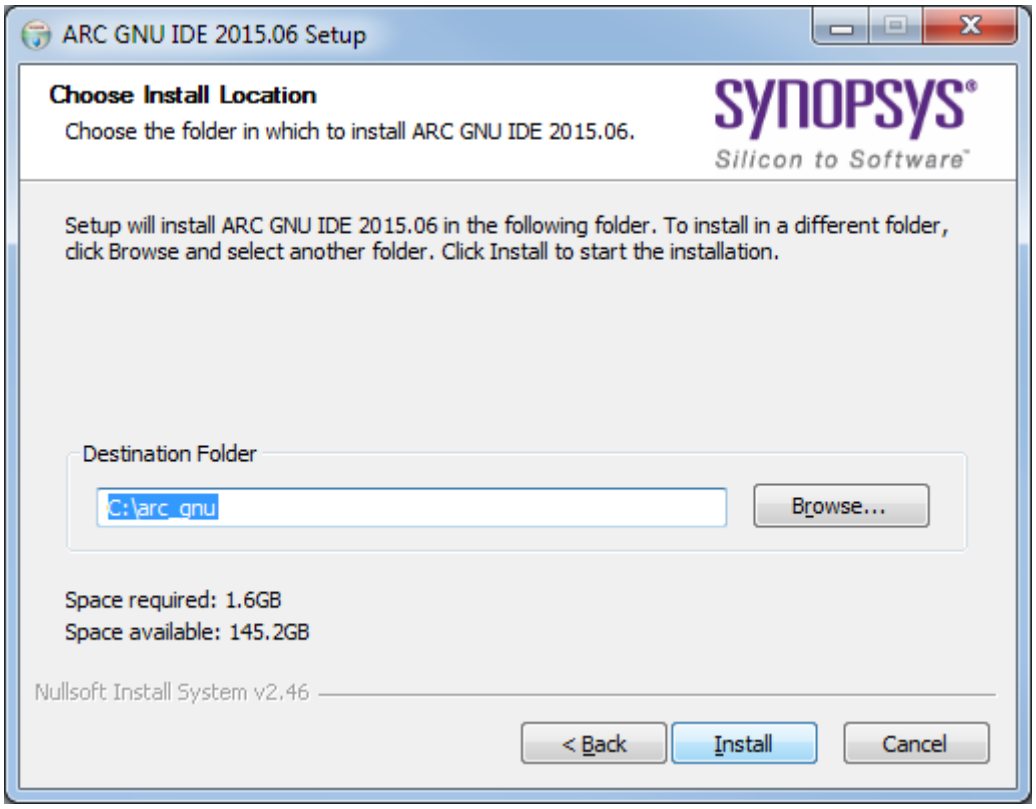
Run arc_gnu_2015.06_ide_win_install.exe



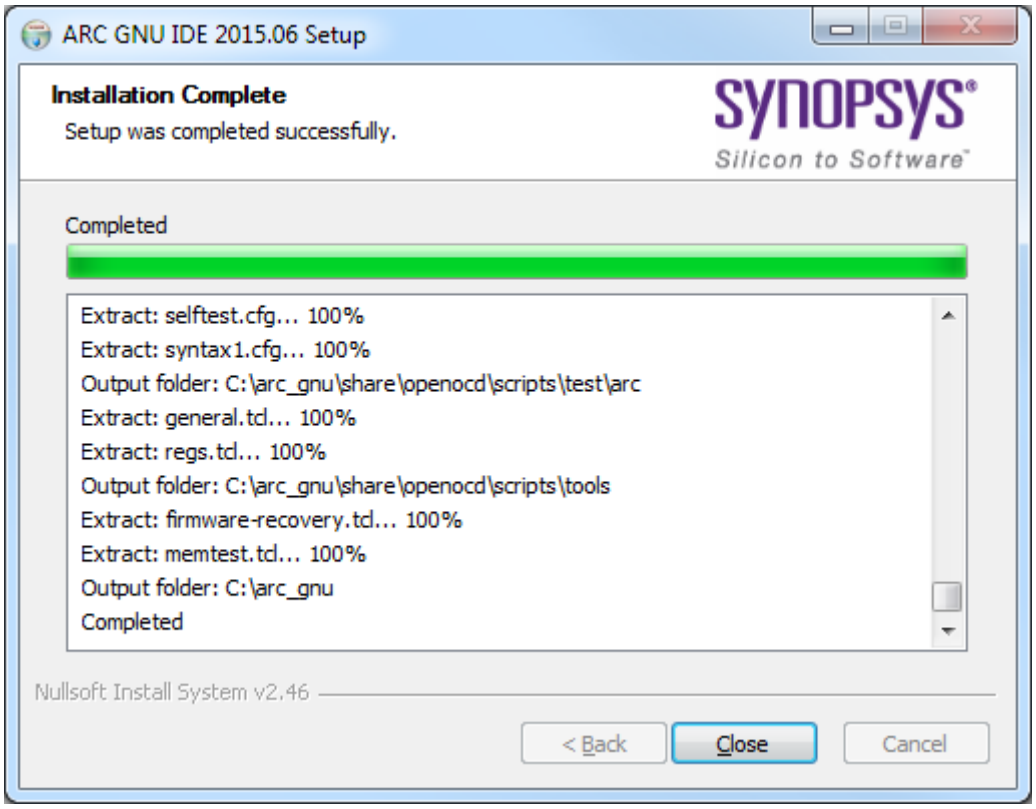
Accept Synopsys FOSS notice



Choose components to be installed



Choose installer paths



Installation Completed

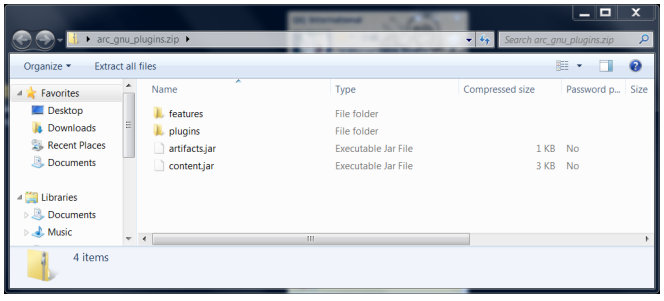
Manual installation on Linux and Windows

Downloading Eclipse

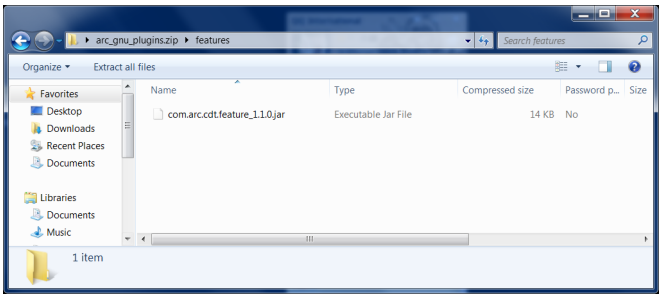
Download Eclipse IDE for C/C++ Developers, that already contains CDT from this page

Downloading latest plugins

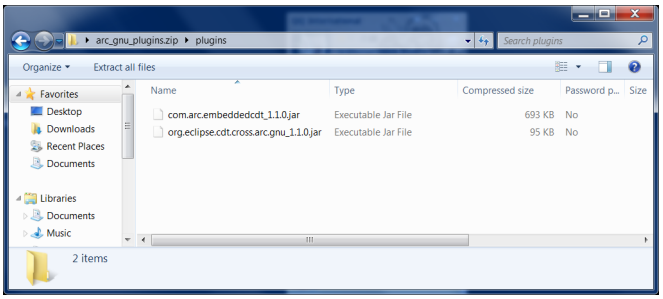
User can get this plug-in from website URL https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/releases/, this is an archived version of the GNU ARC Eclipse plug-in update site, the file name is arc_gnu_<version>_ide_plugins.zip



Components of arc_gnu_ide_plugins.zip



Components of arc_gnu_ide_plugins.zip features

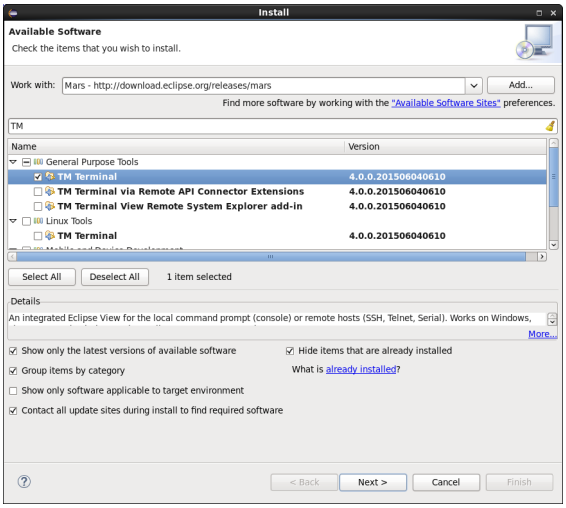


Components of arc_gnu_ide_plugins.zip plugins

Installing into Eclipse

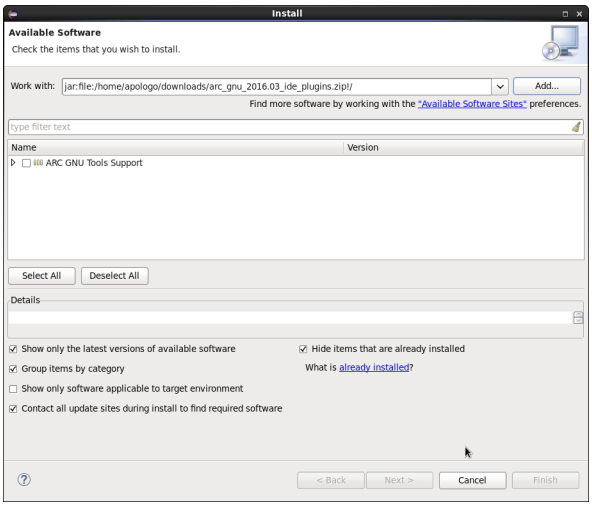
To run ARC plugins for Eclipse, it is required to have Target Terminal plugin installed in Eclipse.

Install “TM Terminal” plugin from Oxygen repository.

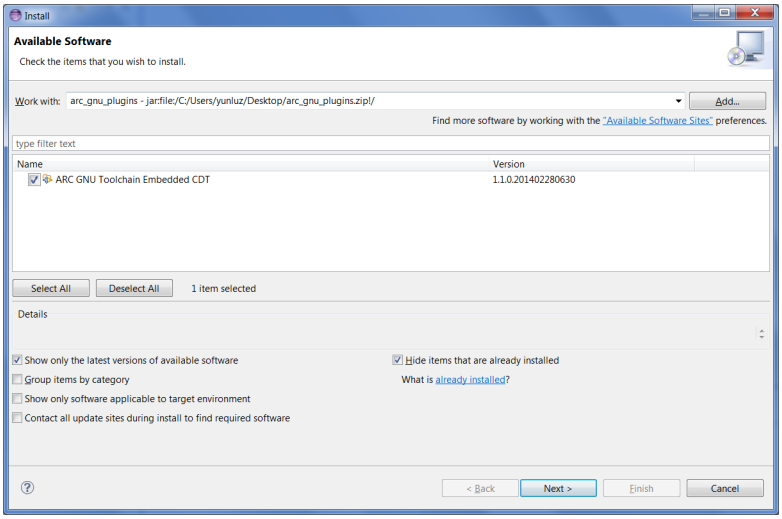


Installation of TM Terminal in Eclipse

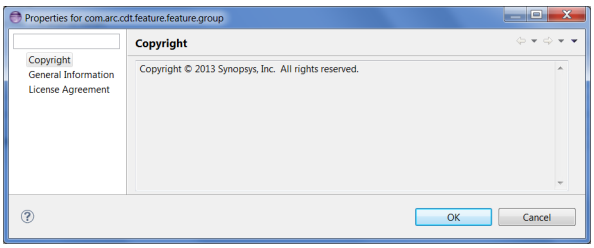
After downloading arc_gnu_ide_plugins.zip successfully, user also can install it from local by pointing Eclipse to it: `Eclipse -> Install New Software -> Add -> Archive ->` select arc_gnu_ide_plugins.zip file. Unzip this archived folder, there will be six components in it.



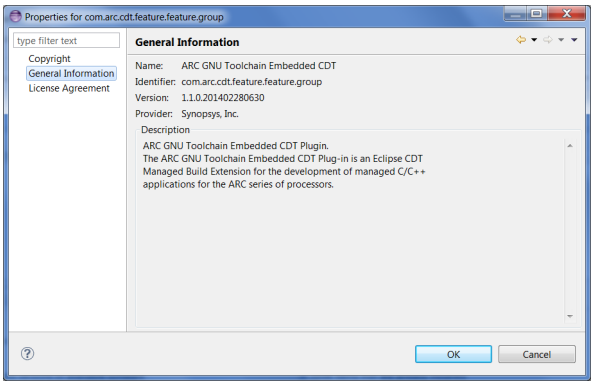
Install from local PC



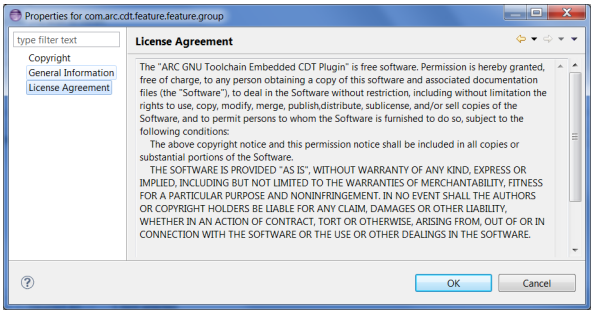
Check GNU ARC C++ Development Support



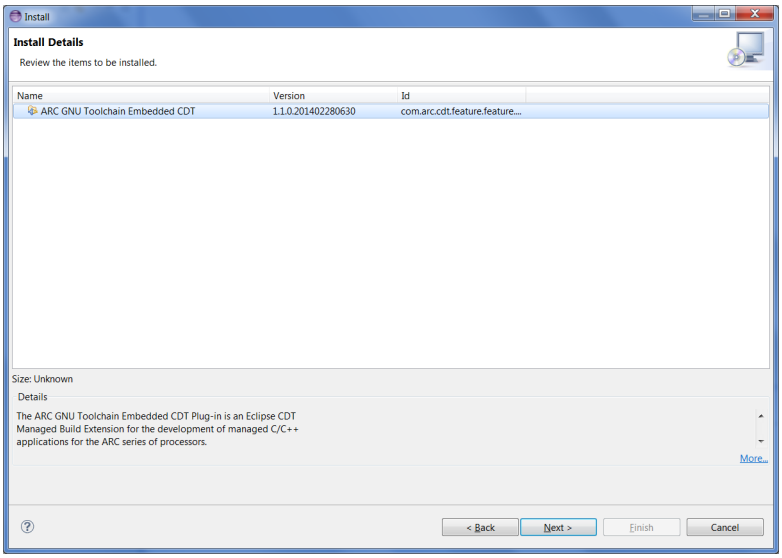
Get copyright by clicking “more”



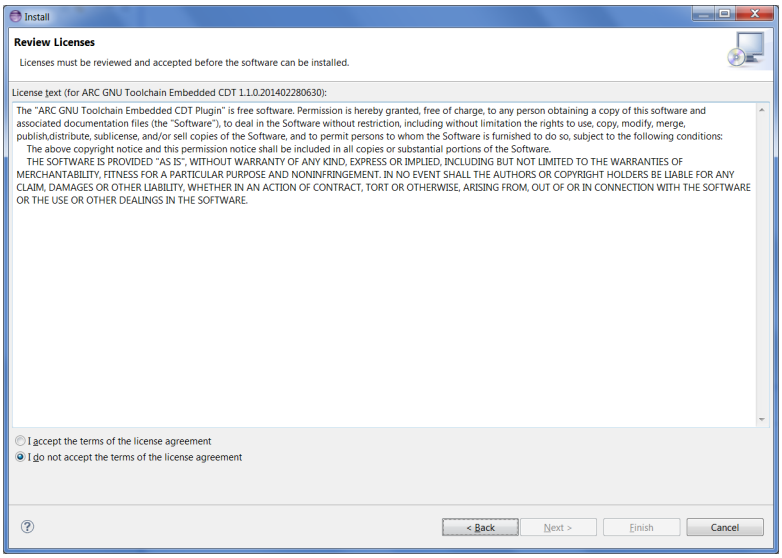
Get General Information by clicking “more”



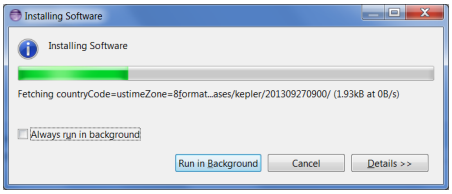
Get License Agreement by clicking “more”



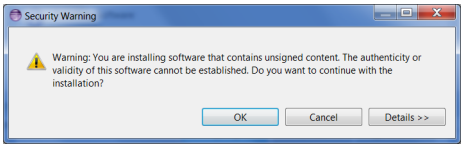
10 Install Details



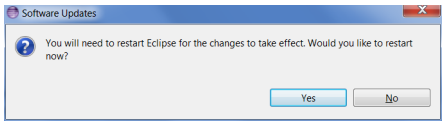
Accept the terms of license agreement



Install ARC GNU IDE Plugin

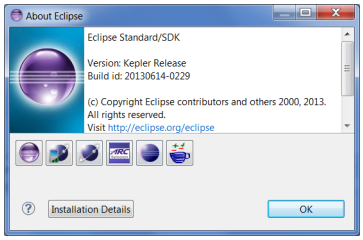


Warning about this plugins installation



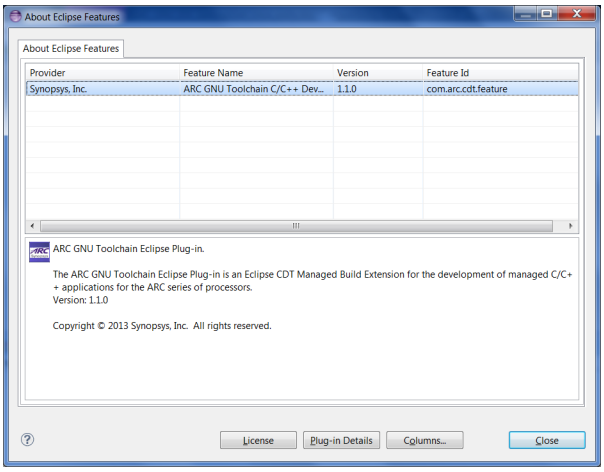
Restarting Eclipse

Ignore the Security Warning, and click “Ok”, after restarting Eclipse IDE, the installation is finished. If user install plug-in successfully, the “ARC” icon will show up in “About Eclipse”.



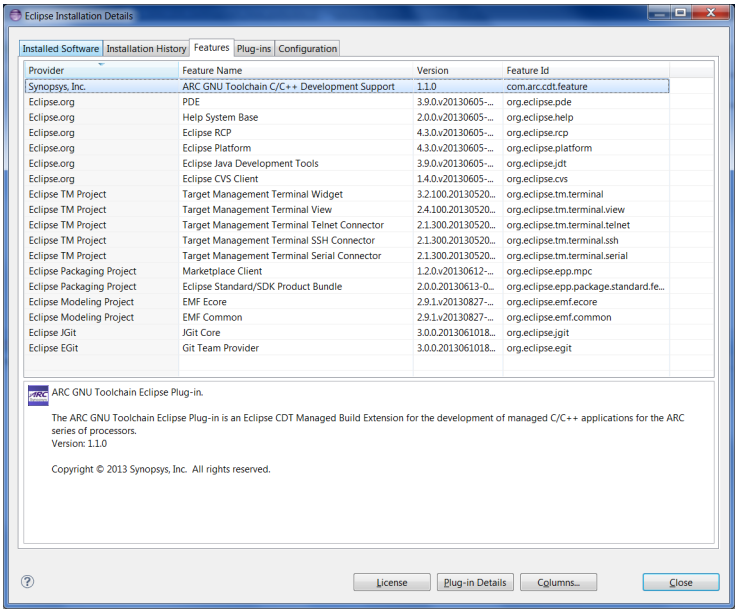
Plug-in in Eclipse IDE

Click the “ARC” icon; user will get detailed plug-in features information.

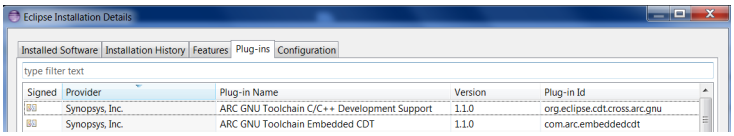


About Eclipse ELF32 Plug-in Features

Click the “Installation Details” button, the Features and Plug-ins will also show up.



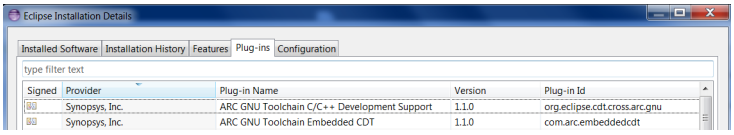
ARC GNU plugin Plug-ins



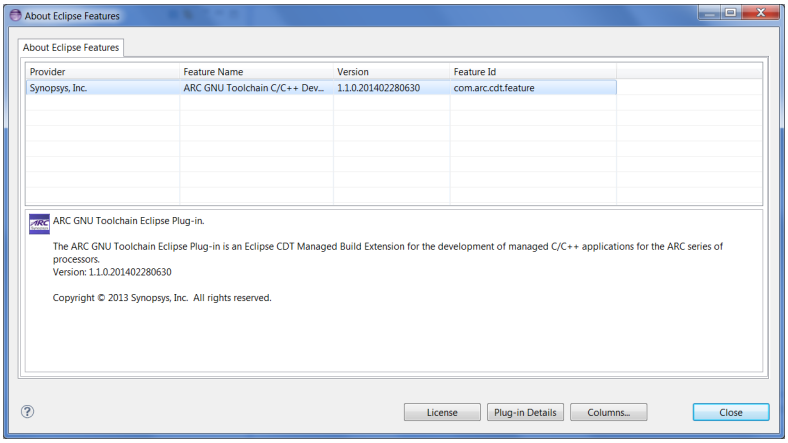
ARC GNU plugin Features

Updating existing plugin

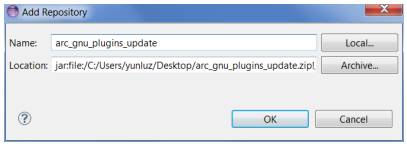
To update the existing plugin, as shown in the figure below, and the version of this current plugin is for example “1.1.0.201402280630”, follow same instructions as plugin installation.



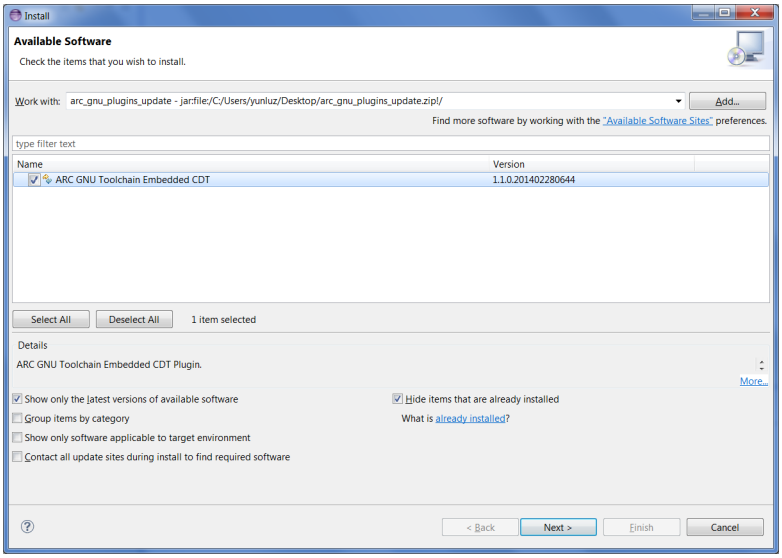
ARC GNU plugin Features



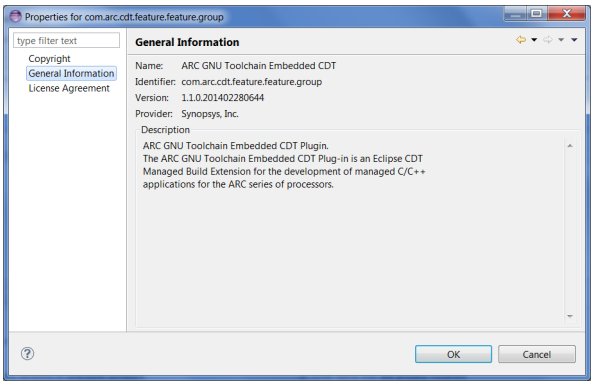
Current ARC GNU IDE plugin



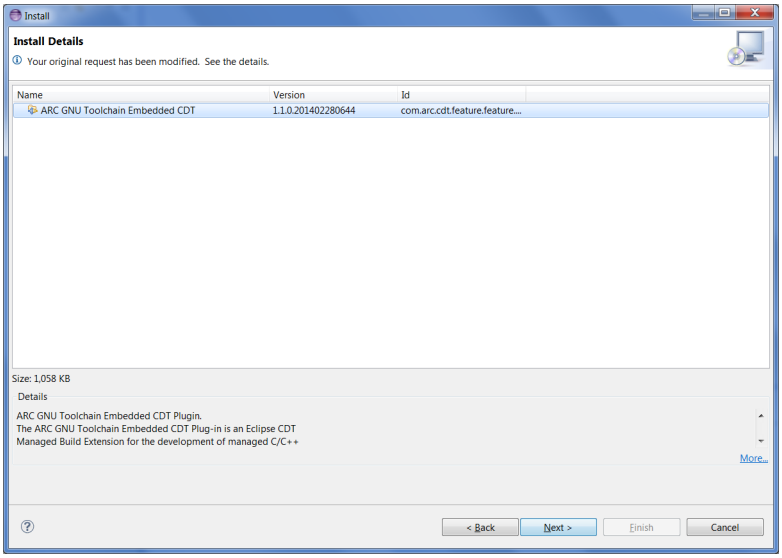
Installation of latest plugin



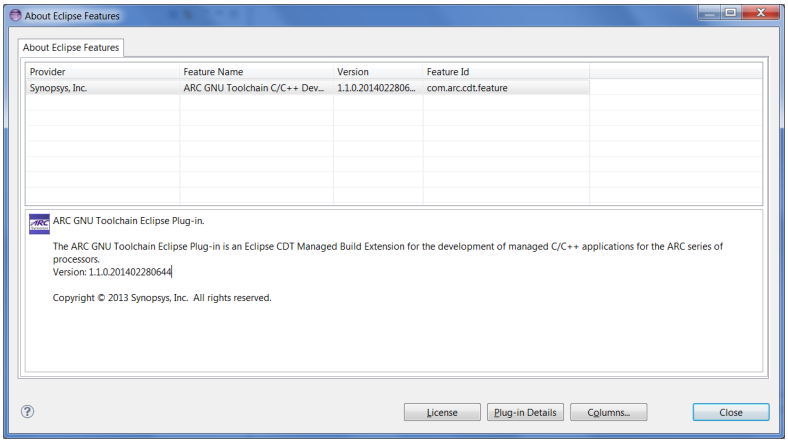
Updated ARC GNU IDE plugin



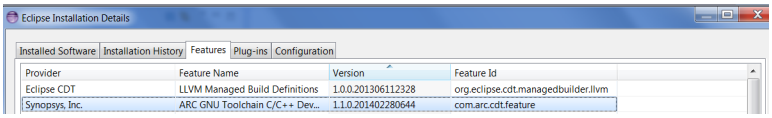
General Information of the latest plugin



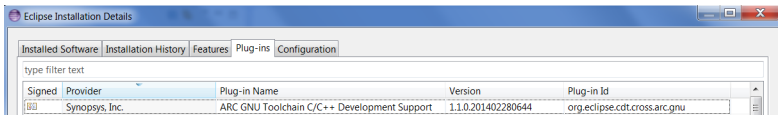
Installed details of the latest plugin



Update exiting plugins successfully



Updated ARC GNU plugin Features



Updated ARC GNU plugin Plug-ins

Installing plugin on Linux host

If you plan to connect to UART port on target board with RxTx plugin controlled by IDE you need to change permissions of directory /var/lock in your system. Usually by default only users with root access are allowed to write into this directory, however RxTx tries to write file into this directory, so unless you are ready to run IDE with sudo, you need to allow write access to /var/lock directory for everyone. Note that if /var/lock is a symbolic link to another directory then you need to change permissions for this directory as well. For example to set required permissions on Fedora:

```
$ ls -l /var/lock
lrwxrwxrwx. 1 root root 11 Jun 27  2013 /var/lock -> ../run/lock
$ ls -ld /run/lock/
drwxr-xr-x. 8 root root 160 Mar 28 17:32 /run/lock/
$ sudo chmod go+w /run/lock
$ ls -ld /run/lock/
drwxrwxrwx. 8 root root 160 Mar 28 17:32 /run/lock/
```

If it is not possible or not desirable to change permissions for this directory then serial port connection must be disable in Eclipse debugger configuration window.

If it is required to connect to UART of a development system, then another problem that might happen is permissions to open UART device. For example on Ubuntu 14.04 only root and members of `dialout` group can use /dev/ttyUSB1 (typical UART port for boards based on FT2232 chip). Thus to use connect to those port user must be made member of `dialout` group. Command to do this:

```
$ sudo usermod -a -G dialout `whoami`
```

If OpenOCD is used, then it is required to set up proper permissions for devices to allow OpenOCD to connect to those devices. Create file `/etc/udev/rules.d/99-ftdi.rulesi` with the following contents:

```
# allow users to claim the device
# Digilent HS1 and similar products
SUBSYSTEM=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", MODE="0664", GROUP="plugdev"
# Digilent HS2
SUBSYSTEM=="usb", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6014", MODE="0664", GROUP="plugdev"
```

Then add yourself to `plugdev` group:

```
$ sudo usermod -a -G plugdev `whoami`
```

Then restart udev and relogin to system, so changes will take effect.:

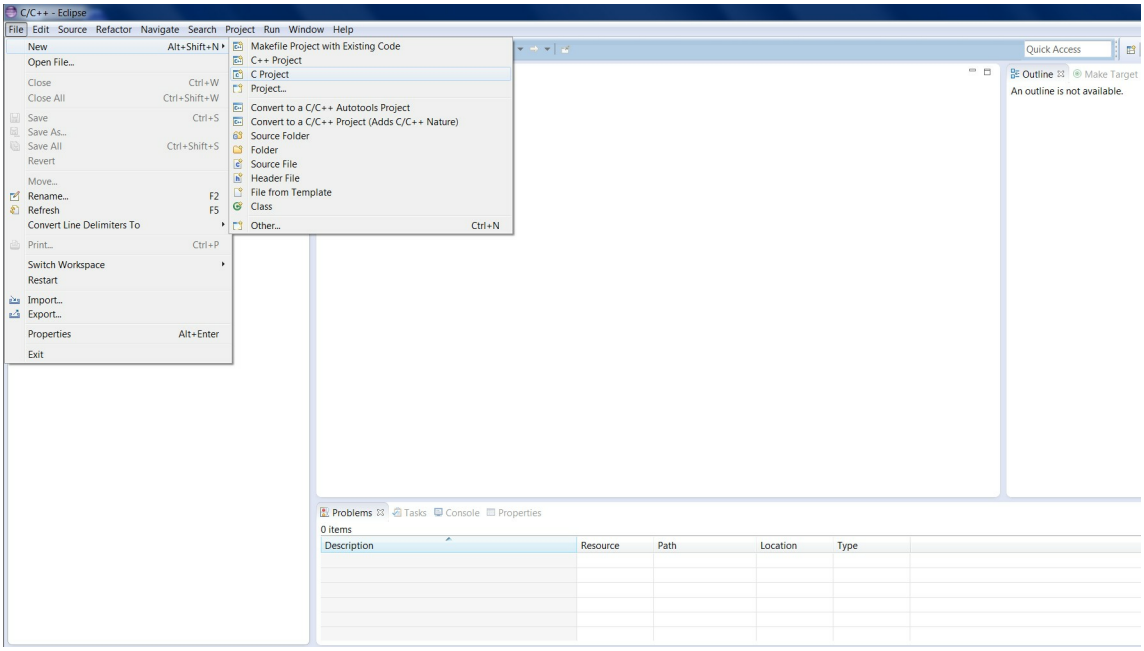
```
$ sudo udevadm control --reload-rules
# Disconnect JTAG cable from host, then connect again.
```

Even better is to reboot the system.

Getting Started with EM Starter Kit

Creating Hello World project for ARC EM Starter Kit

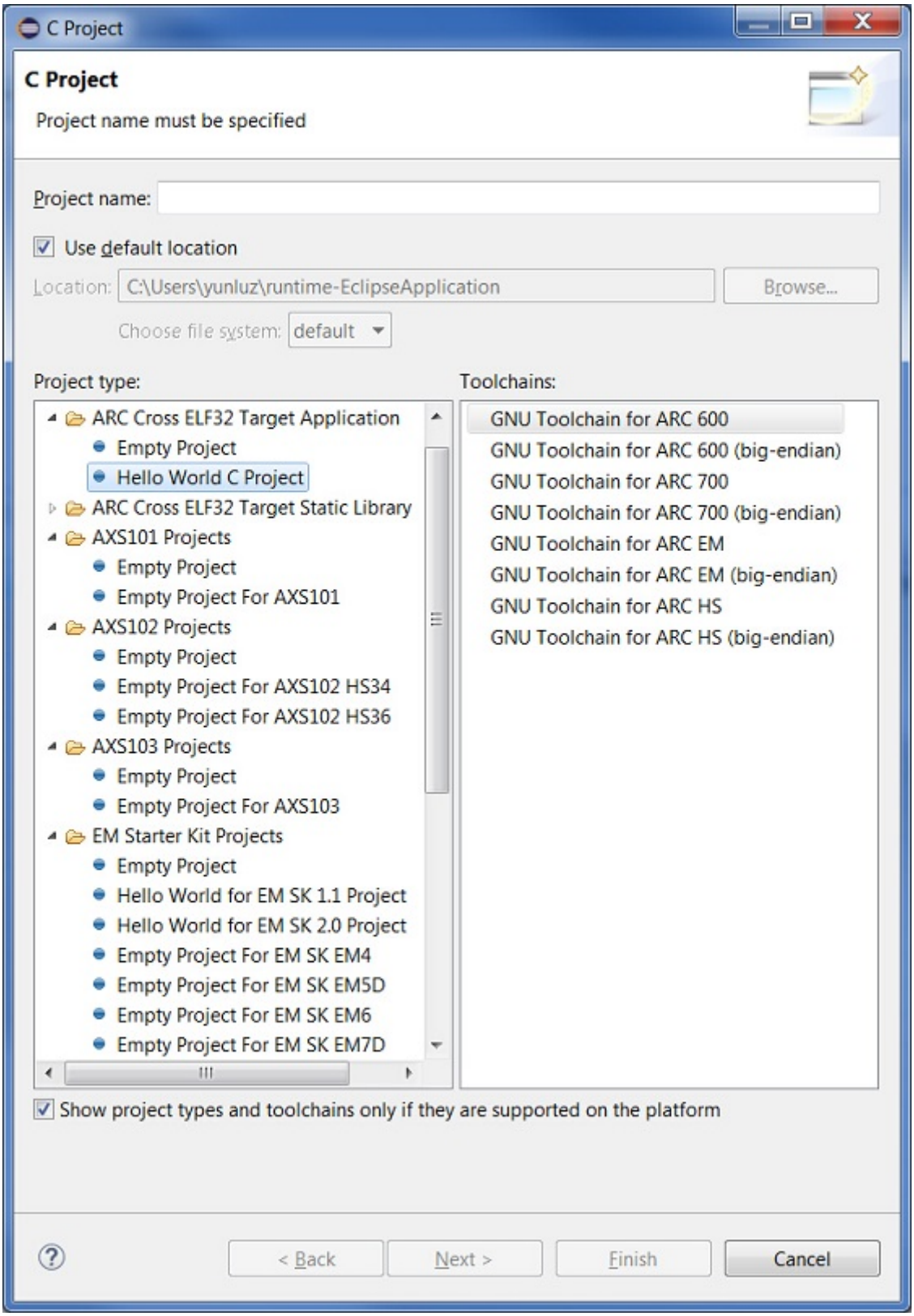
1. Select **File >New >C Project**



Creating a new C project

The **C Project** window should have several ARC project types: **ARC Cross ELF32 Target Application**, **ARC Cross ELF32 Target Static Library**, **AXS10x Projects** and **EM Starter Kit Projects**. Select **EM Starter Kit Projects**. This project type is available only if ARC EM toolchain compiler can be found either in `PATH` environment variable or in `../bin/` directory relative to Eclipse executable.

2. Enter a project name

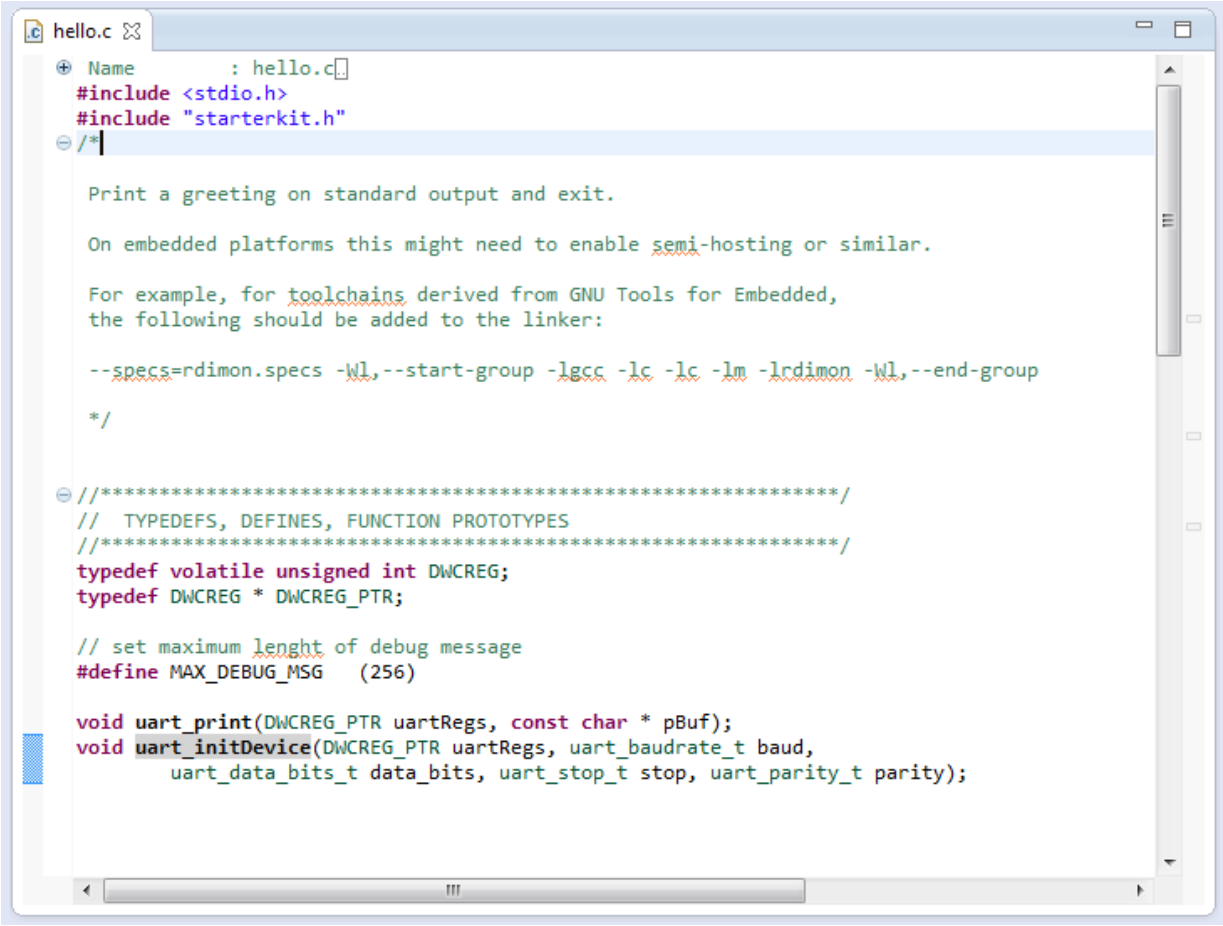


C Project Creation Dialog

1. Under **EM Starter Kit Projects** select **Hello World for EM SK 2.1 Project** (or **Hello World for EM SK 1.1** depending on your EM Starter Kit version). This Hello World project is an example project that comes with the EM Starter Kit Software and uses the UART on the EM Starter Kit to display output. To see output printer to UART, connection to serial port should be established. By default Eclipse for ARC will automatically connect to available serial port, but that feature can be disabled and in that case refer to the EM Starter Kit Getting Started for instructions on how to connect to the UART on the EM Starter Kit board using Putty.

2. Fill in project name and click **Finish** or **Next** if you want to fill in additional information.

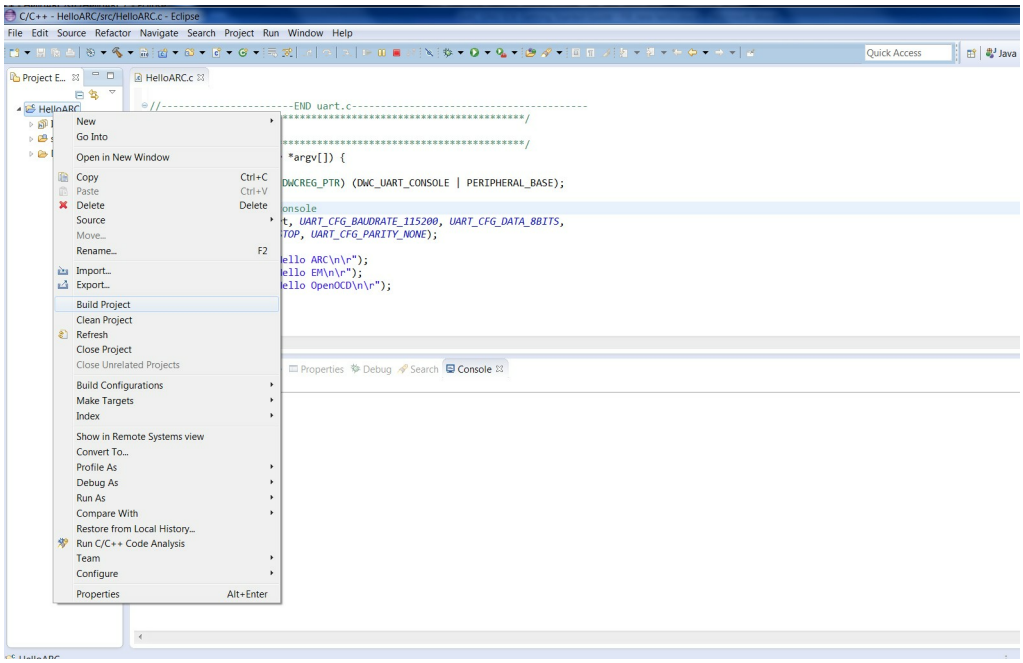
The resulting Hello World project created in Eclipse is:



Final Hello World Project in Eclipse

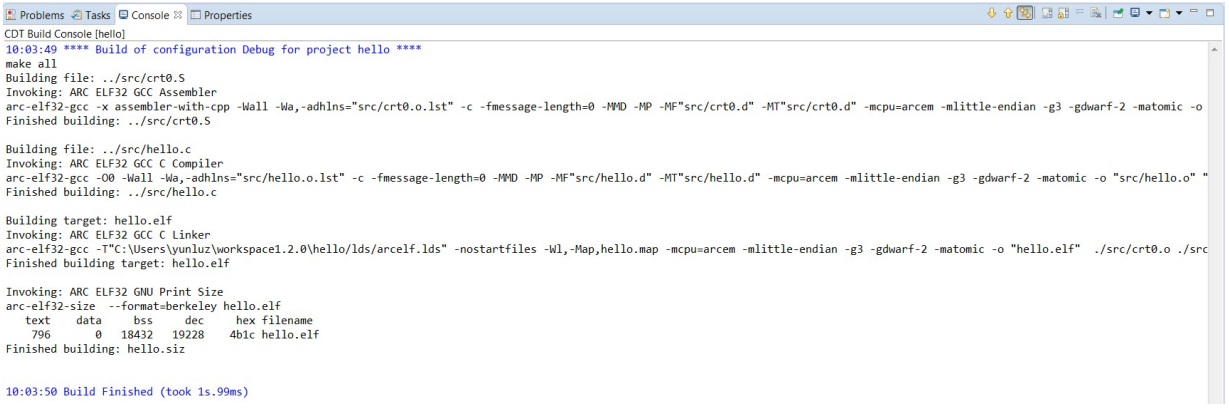
Building the project

1. Right click on the *Hello World project* and select **Build Project** from the pop-up menu



Building a Project

2. Review the build output log in the Eclipse console tab to confirm success:



Build Output

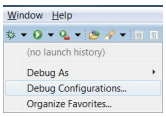
Debugging the project

Once the project is successfully compiled by ARC GCC, you can debug the resulting executable on the EM Starter Kit board.

> **Note**
>
> If you are using Windows platform, please configure drivers for your EM Starter Kit before you begin debug process. For the instructions see How to use OpenOCD on Windows.

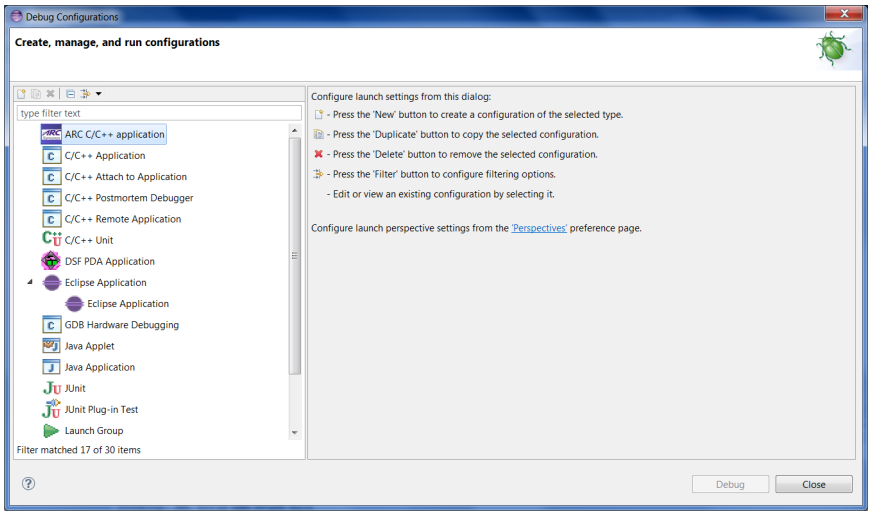
To debug the project, create a new debug configuration.

1. Select **Debug Configurations** from the **Run** menu or by clicking on the down arrow next to the bug icon:



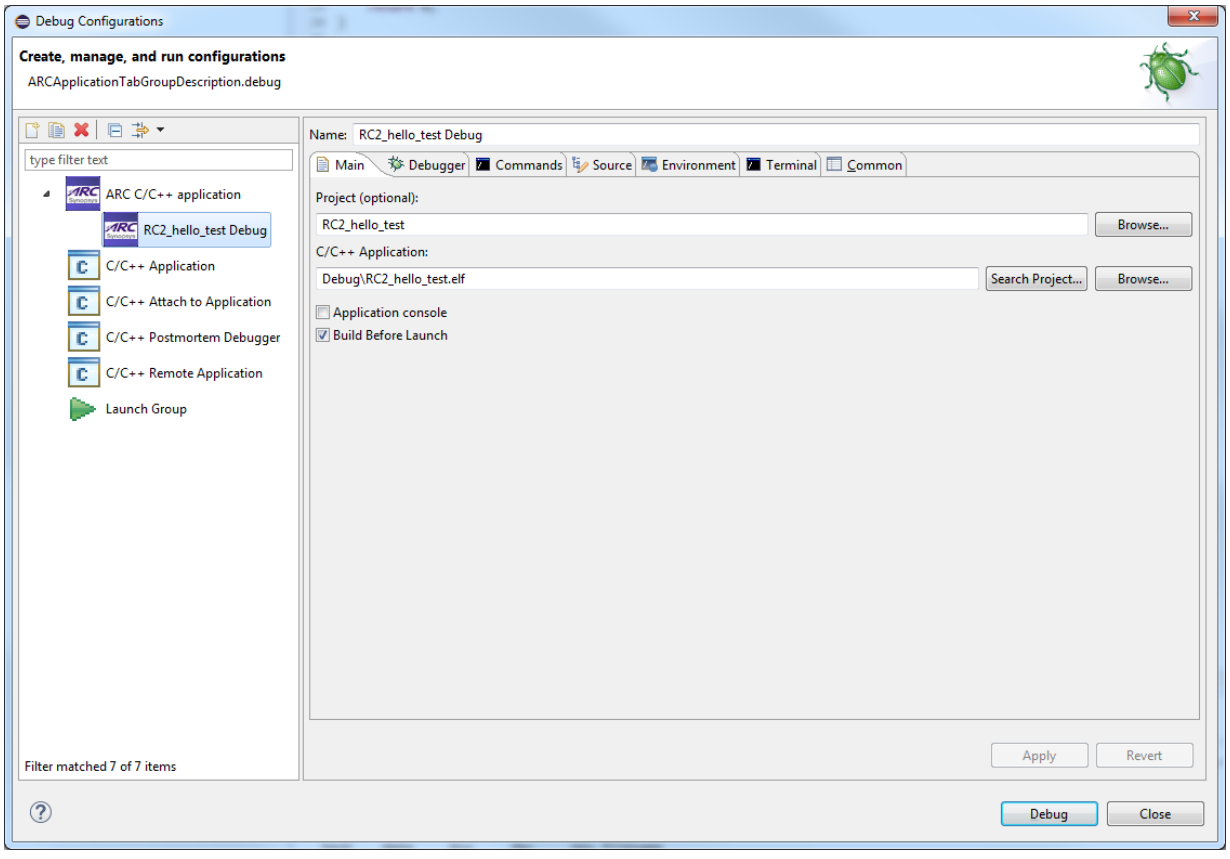
Debug Configurations

2. Double click on the **ARC C/C++ Application** or click on the top left icon to create a new debug configuration for the project:



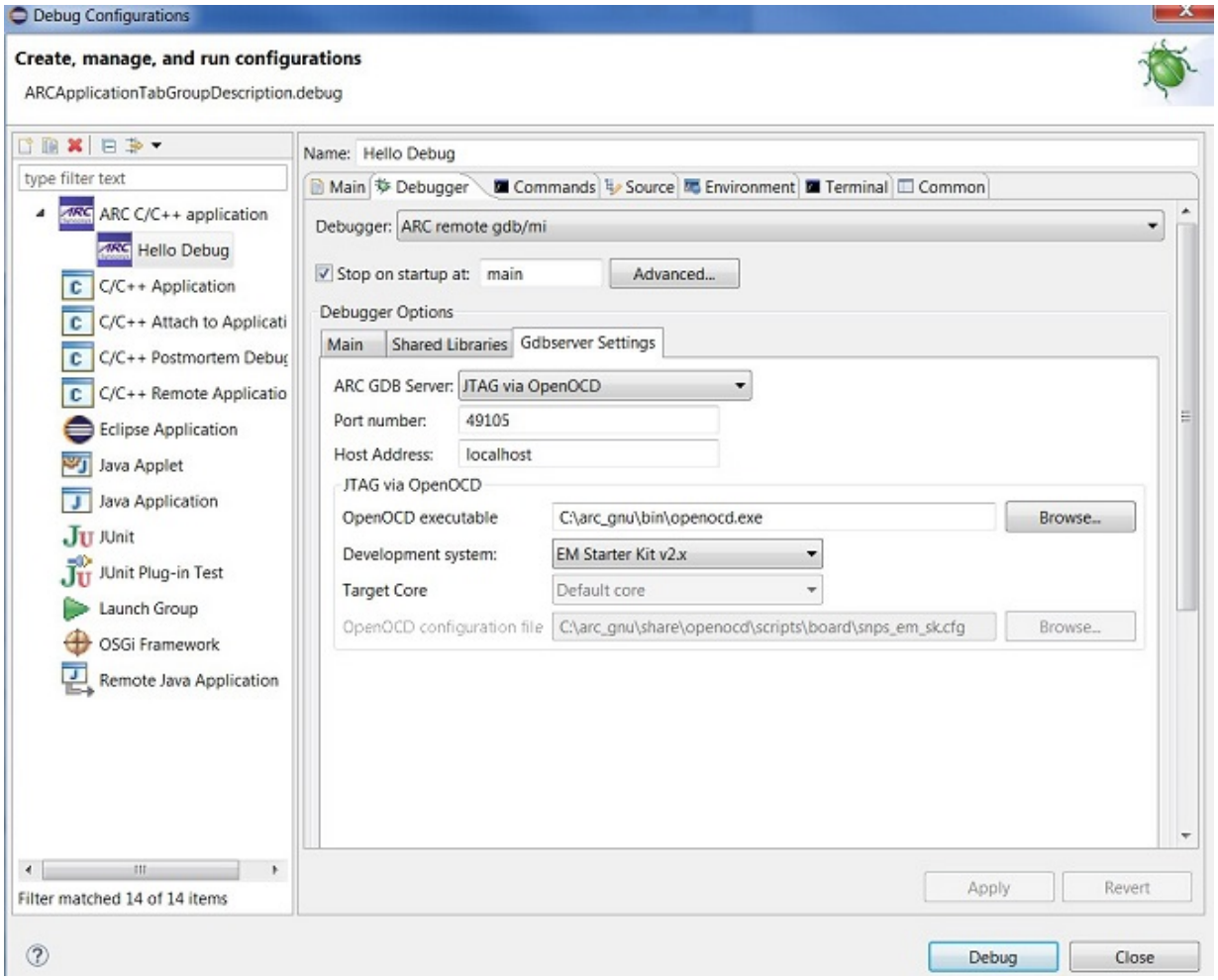
ARC Embedded Debug Configurations

3. Select a name for the new debug configuration (by default, it equals the project name followed by “Debug”).



New debug Configuration

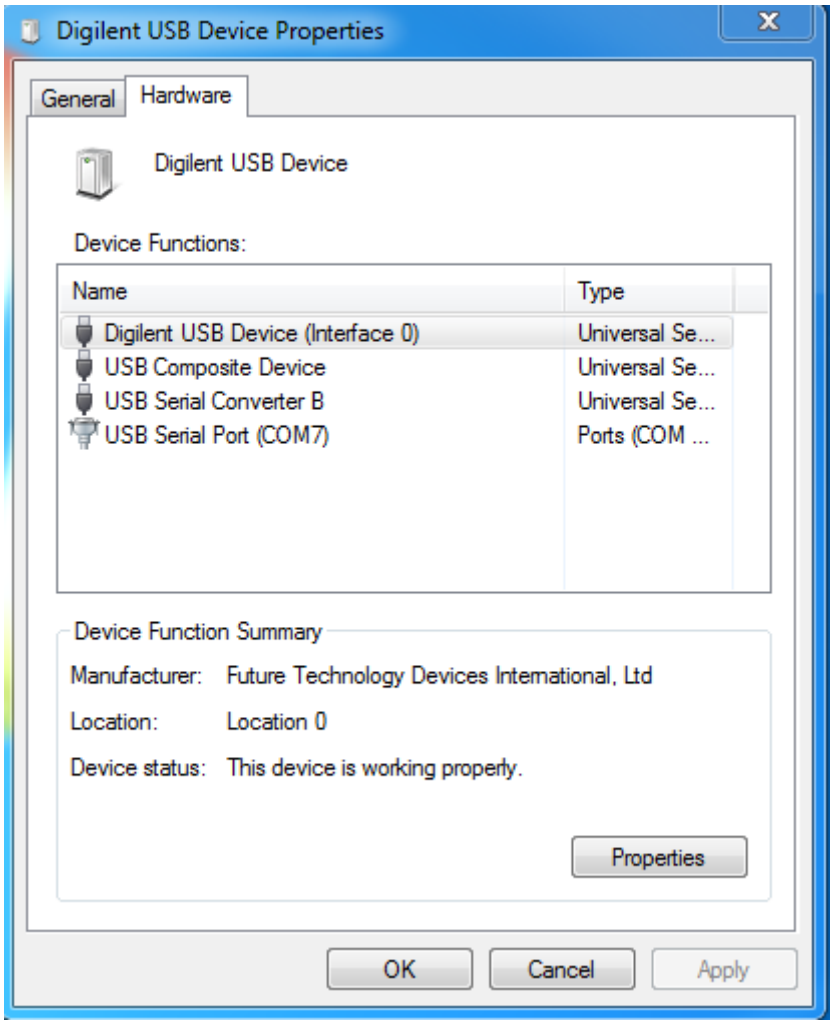
4. Click the **Debugger** tab and select **Gdbserver Settings** page.



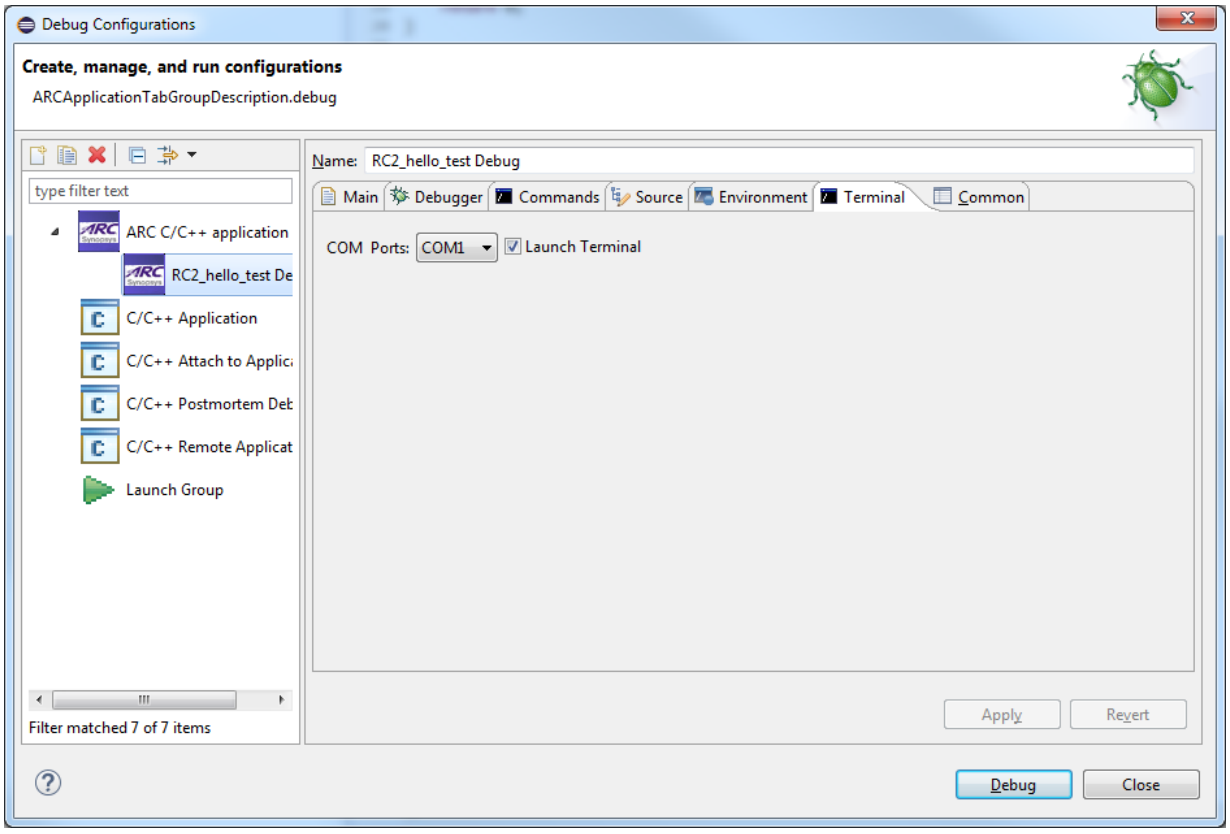
Default values in the Debugger tab for JTAG via OpenOCD

Select **EM Starter Kit v2.x** or **EM Starter Kit v1.x** as development system depending on your EM Starter Kit version.

Open **Terminal** tab and select COM Port. For Linux choose **/dev/ttyUSB1**. For Windows COM Port should match the port number in “Device and Printers” as shown below.



Digilent USB Serial COM Port

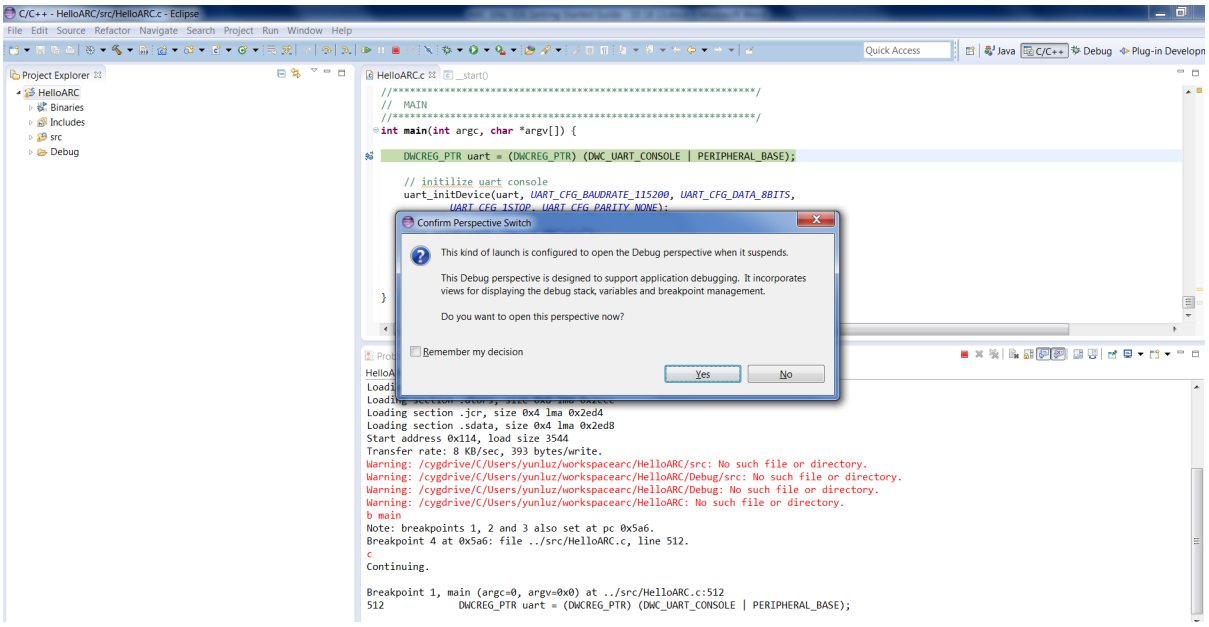


USB Serial COM Port

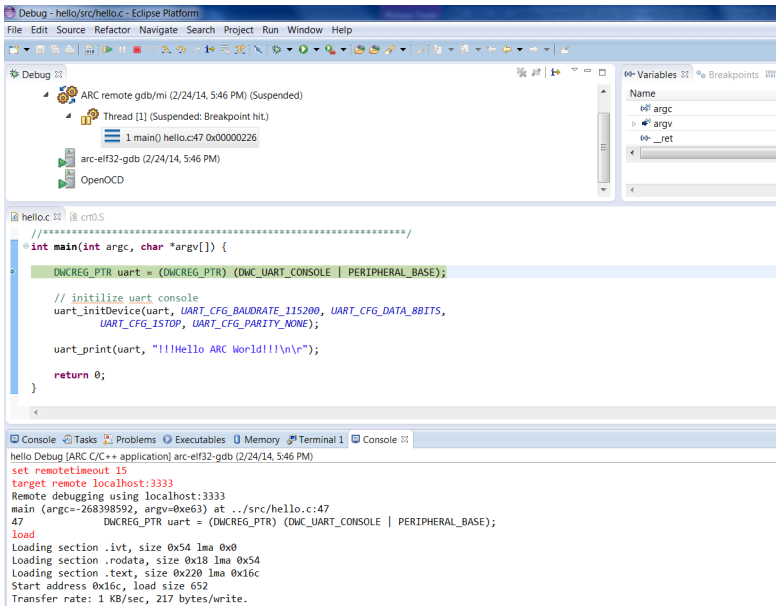
5. Click the **Debug** button in the **Debug configurations** dialog to initiate debug session.

   This action automatically launches the Serial terminal and OpenOCD applications in the background and connects to the UART on the EM Starter Kit board.

6. Click **Yes** in the confirmation dialog to switch to the Debug perspective



Perspective Switch



Debugging Process

Getting Started with nSIM

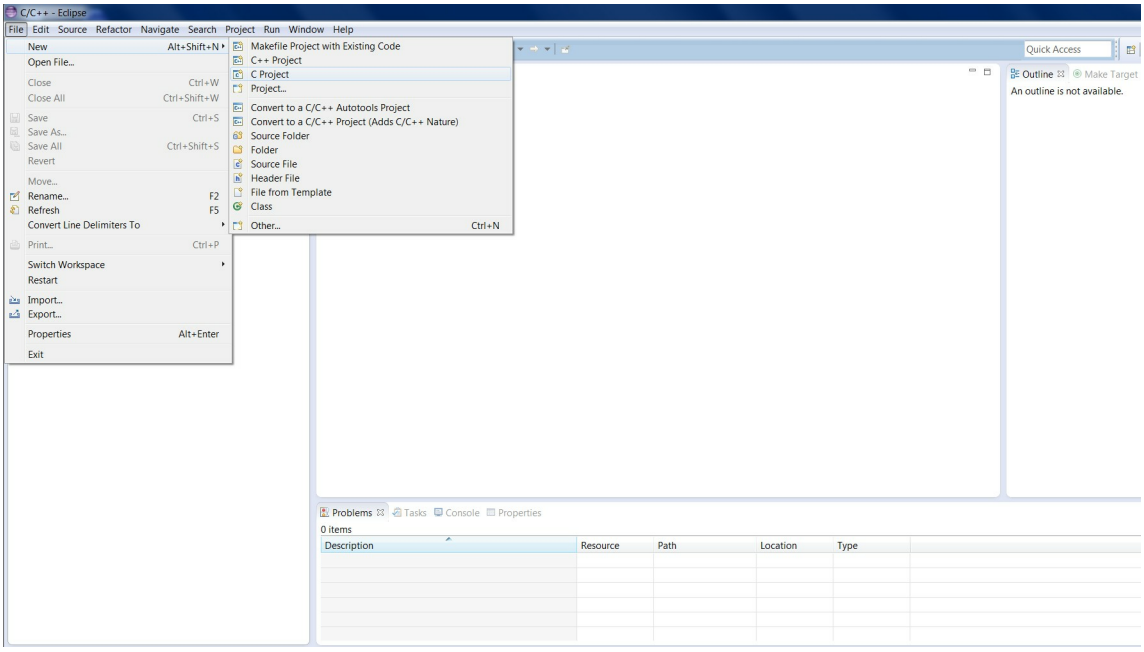
Prerequisites

You should have nSIM installed on your computer.

You also might need to set environment variable **LM_LICENSE_FILE** =<your_license> in case if you have full nSIM version. Otherwise you will get licensing failure.

Creating a C project

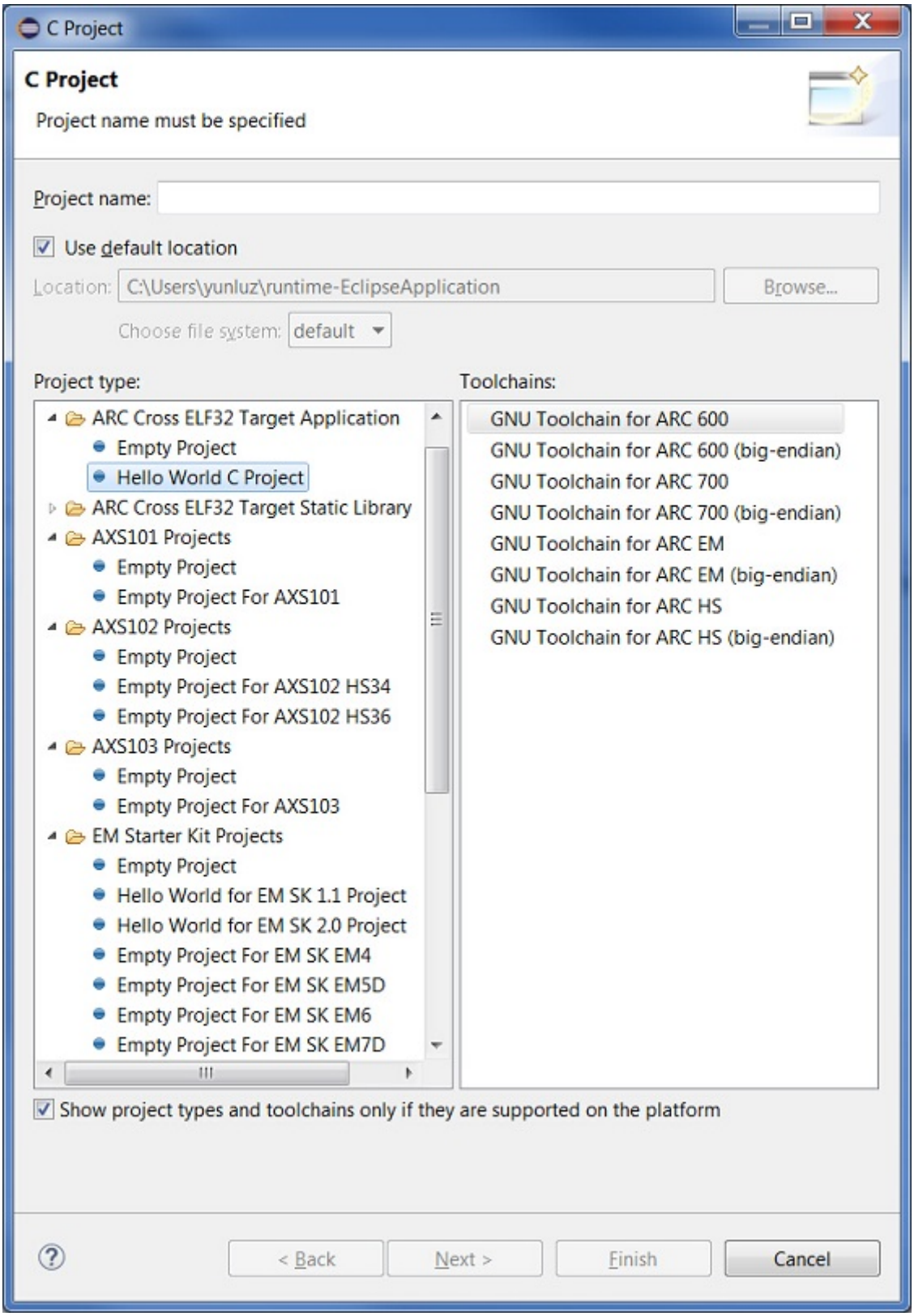
1. Select **File >New >C Project**



Creating a new C project

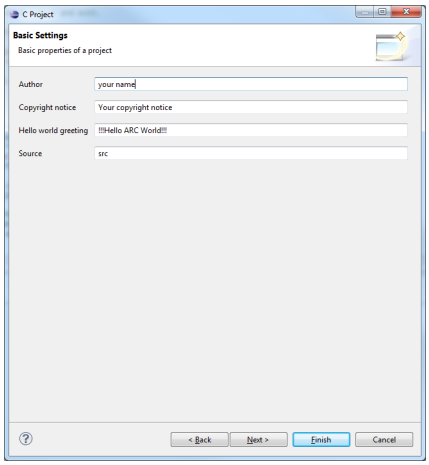
In **C Project** window select **Hello World C Project** under **ARC Cross ELF32 Target Application** project type. On the right side of the window there is a list of available toolchains, select **GNU Toolchain for ARC EM**. If you do not see this toolchain in the list or this project type in the list of project types, make sure that ARC EM toolchain compiler is in the `PATH` environment variable or at `../bin/` directory relative to Eclipse executable.

2. Enter a project name



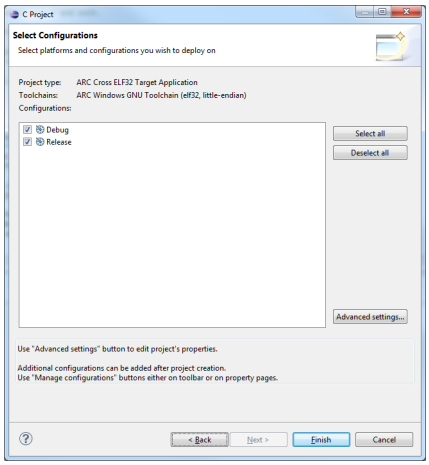
C Project Creation Dialog

After that you can click **Finish** or you can click **Next** and fill in additional information about your project.



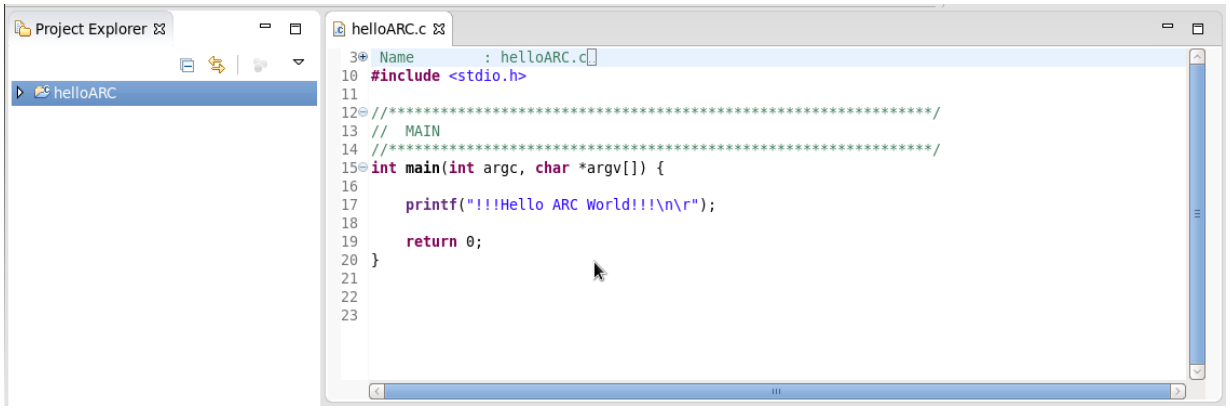
C project creation: Additional information Dialog

Select the desired configuration and click **Finish**.



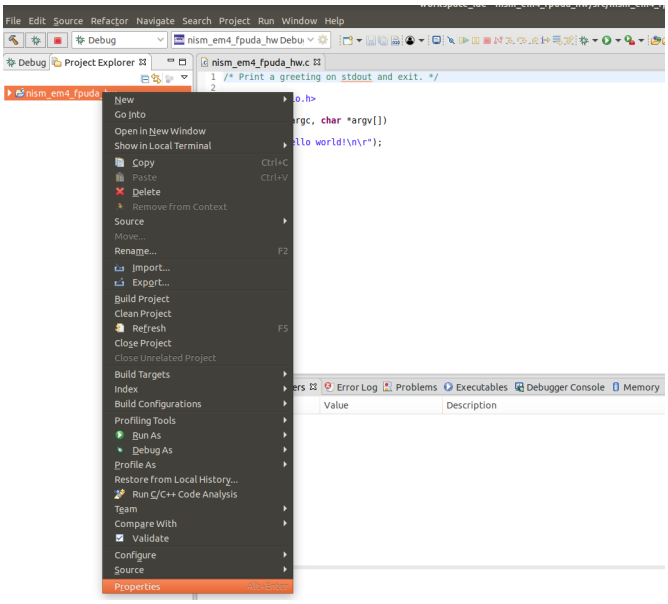
C Project creation: Configurations Dialog

The resulting Hello World project created in Eclipse is:



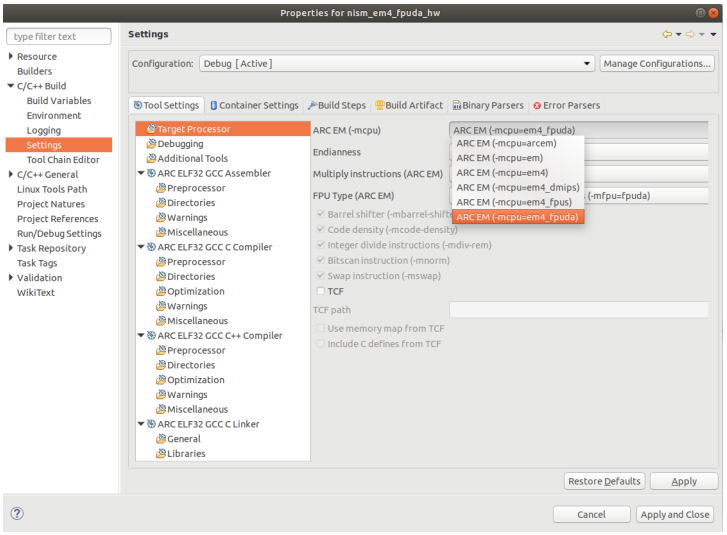
C Project for nSIM debugging

To change -mcpu option: Right click on project and select **Properties** (or **Alt + Enter**)



C Project Properties

Then click **C/C++ Build** -> **Target Processor** and choose required option option



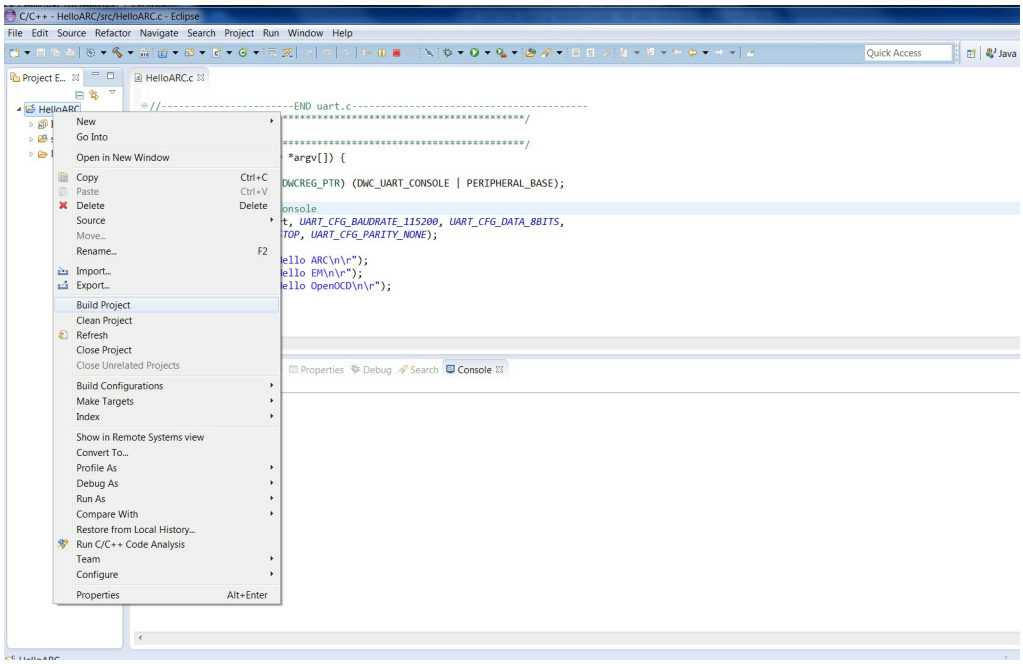
C Project -mcpu options

Building the project

Note

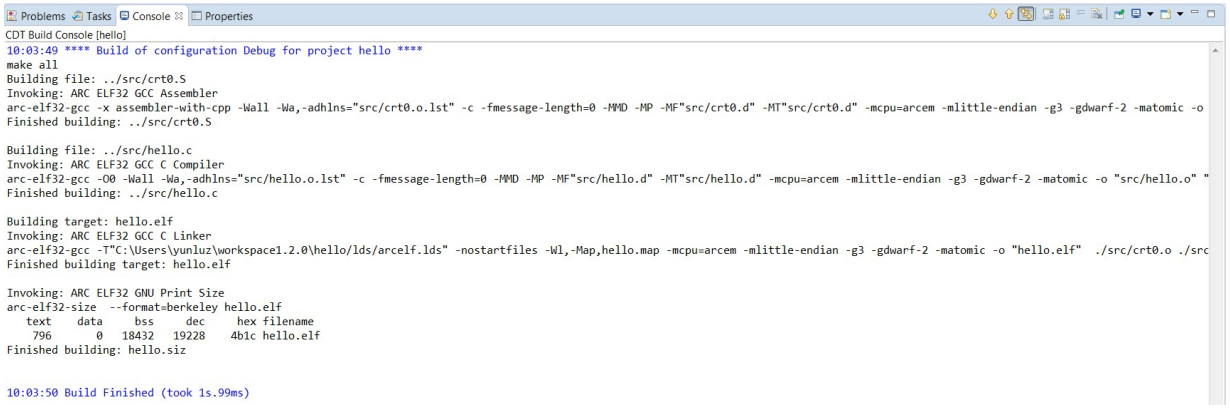
Simple “Hello world” application may be too big to fit into ICCM of a typical ARC core, because `printf()` function pulls in a lot of infrastructure which is usually not present in embedded applications. For demonstration purposes it is recommended to use TCF templates with an external memory instead of CCM. Otherwise there would be a warning from the linker during application build and application may not run correctly on nSIM. So, to run “Hello world” example it is not recommended to use any of `em4*`, `em5*`, `em9*`, or `hs34*` templates.

1. Right click on the *Hello World project* and select **Build Project** from the pop-up menu



Building a Project

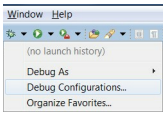
2. Review the build output log in the Eclipse console tab to confirm success:



Build Output

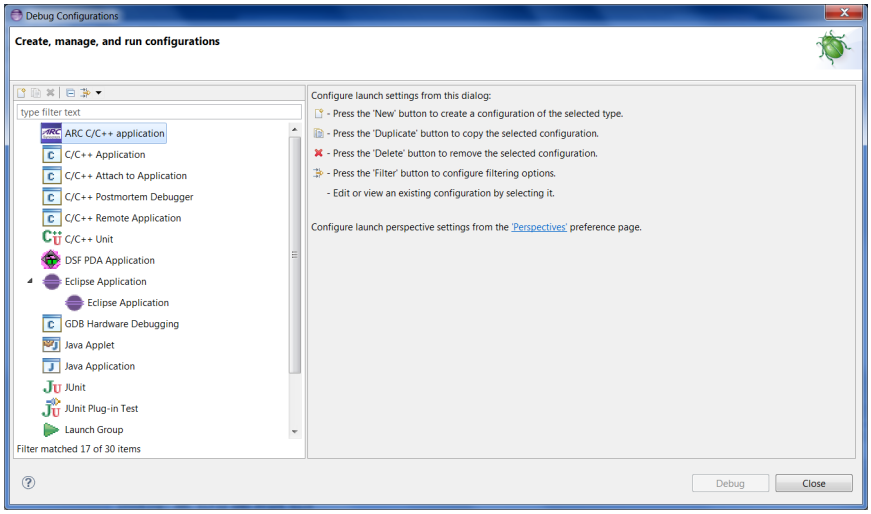
Debugging the project

1. Select **Debug Configurations** from the **Run** menu or by clicking on the down arrow next to the bug icon:



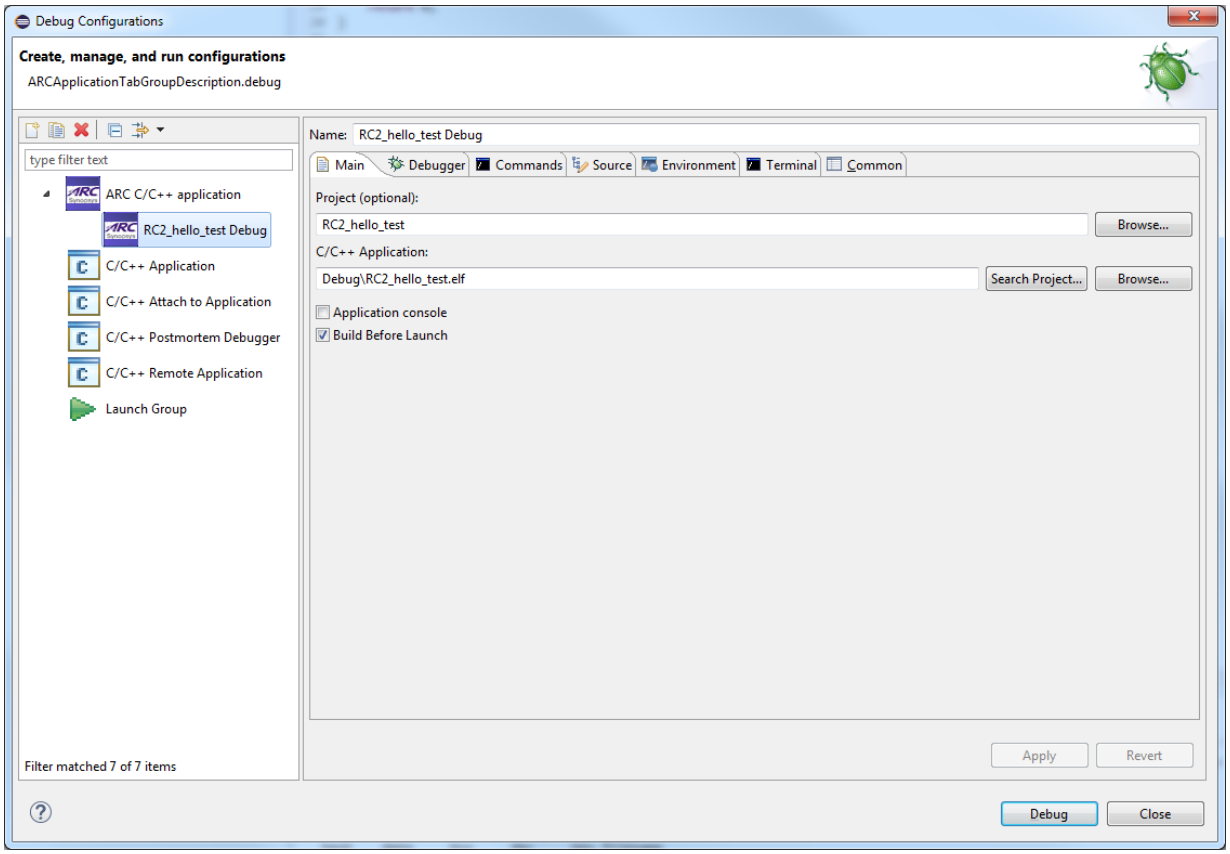
Debug Configurations

2. Double click on the **ARC C/C++ Application** or click on the top left icon to create a new debug configuration for the project:



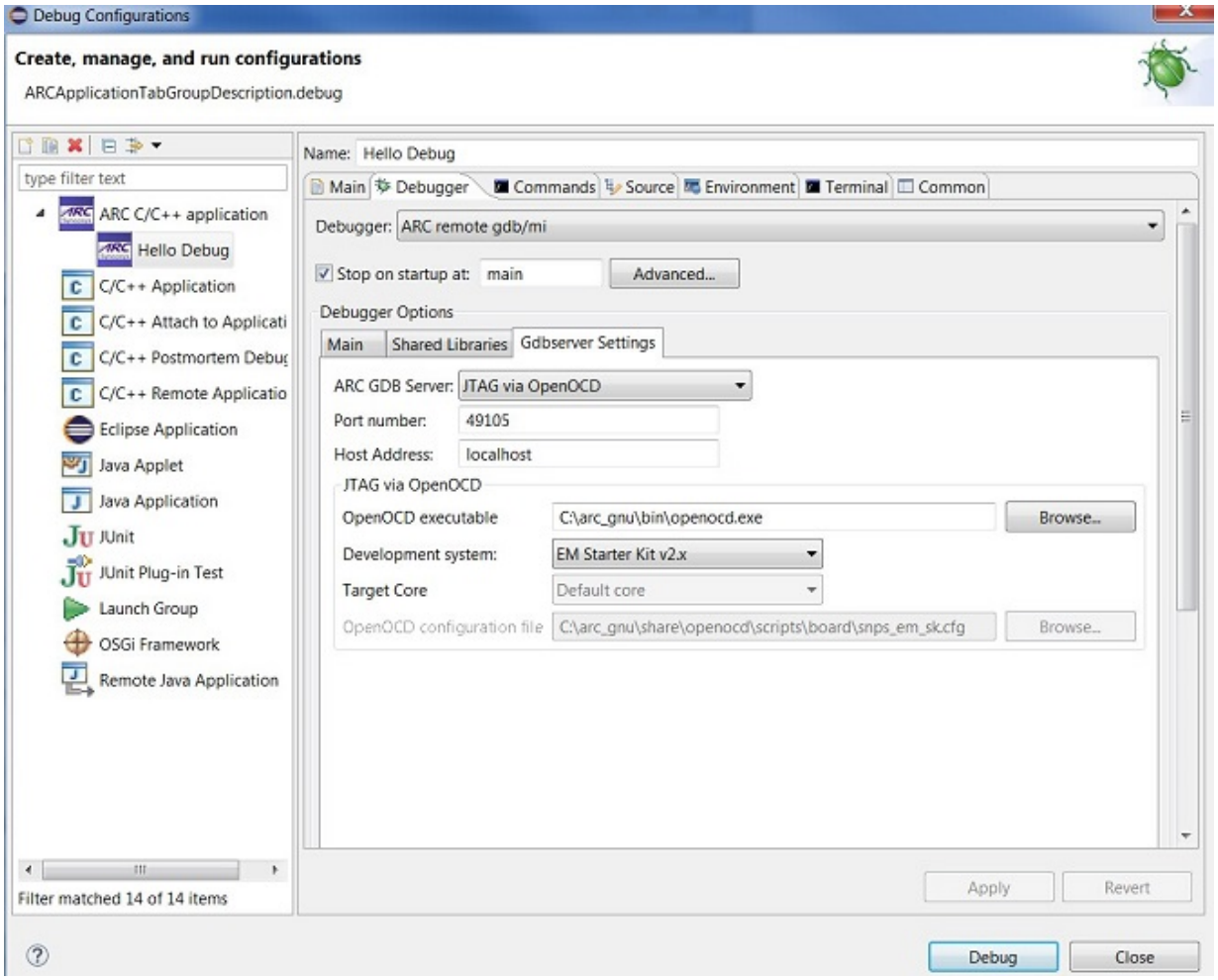
ARC Embedded Debug Configurations

3. Select a name for the new debug configuration (by default, it equals the project name followed by “Debug”).



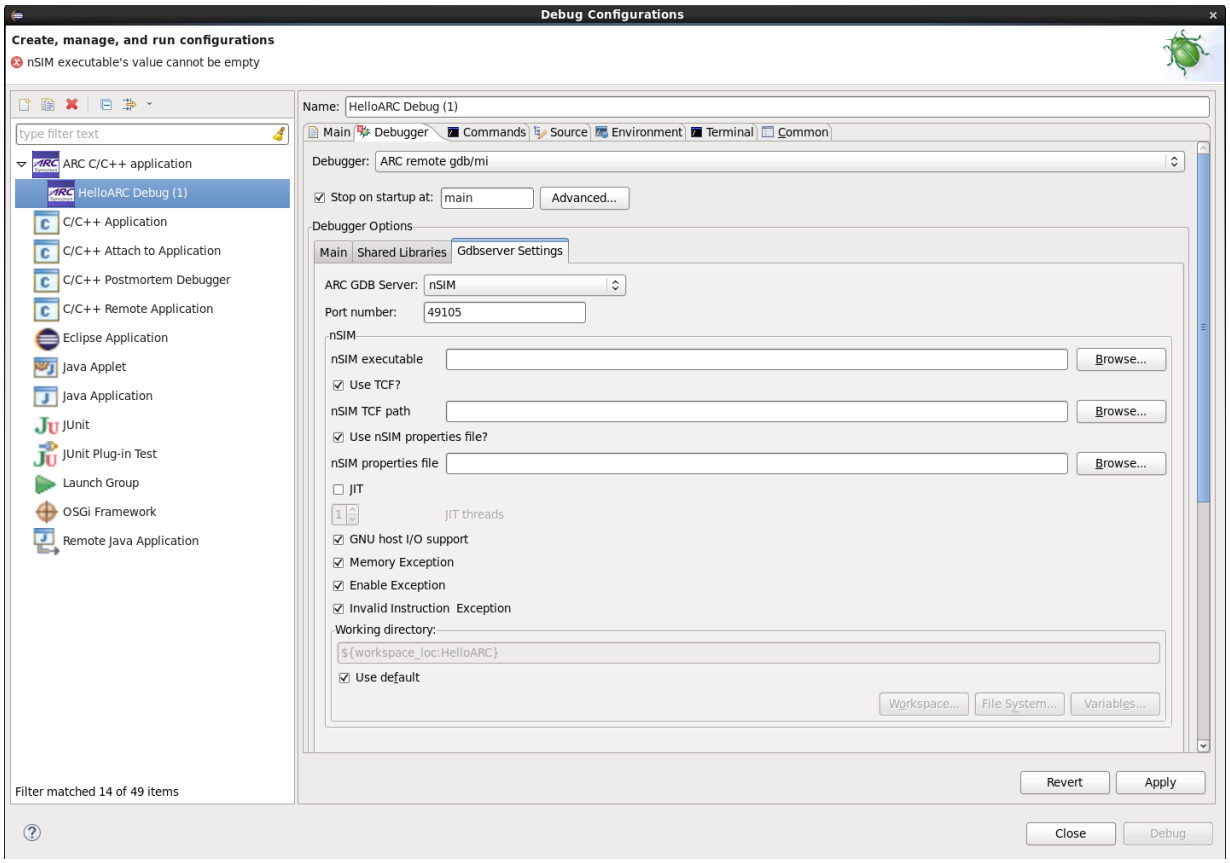
New debug Configuration

4. Click the **Debugger** tab and select **Gdbserver Settings** page.



Default values in the Debugger tab

Select **nSIM** in the **ARC GDB Server** droplist.

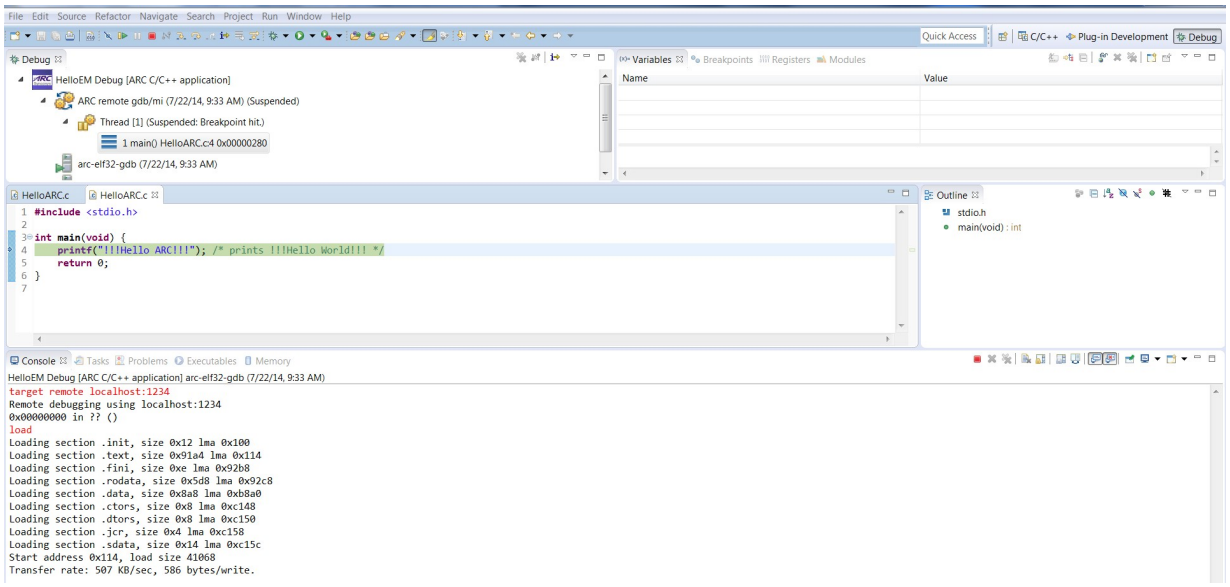


Choosing nSIM on debug tab

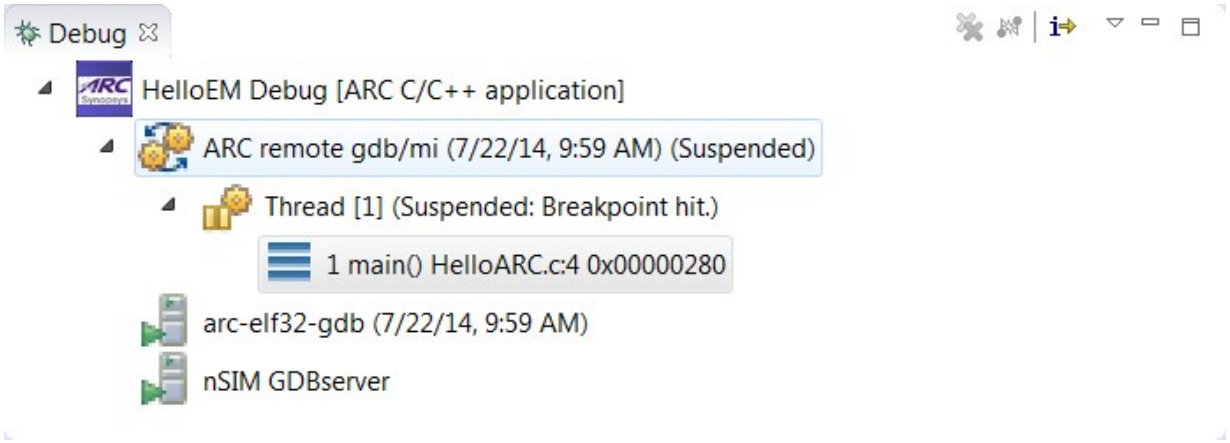
In this tab you should specify paths to nSIM executable and to a TCF file. nSIM has several TCF file templates in the folder `../etc/tcf/templates` relative to nSIM executable file. Choose *em6_dmips.tcf* file from templates. Then uncheck **Use nSIM properties file?** checkbox and click **Apply** button.

If you choose big endian toolchain, then you need to change .tcf file because it is configured for little endian case. Open your .tcf file and find **nsim_isa_big_endian** option in nsim.props field. Set it's value to **1**. This way nSIM will be configured for big endian.

5. To debug application using nSIM, press "Debug" button of IDE.



Debugging with nSIM gdbserver



Debug Window

nSIM gdbserver output in console

Creating and building an application

ARC Project Templates

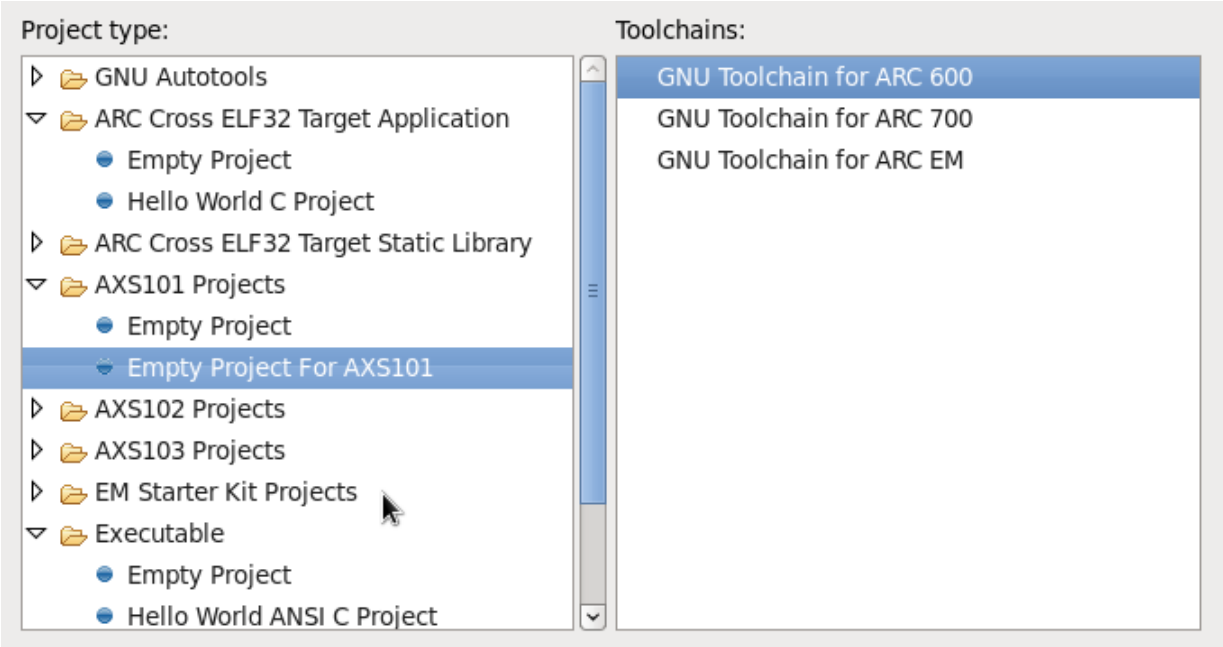
There are several ARC projects types available in the C project dialog: **ARC Cross ELF32 Target Application**, **ARC Cross ELF32 Target Static Library**, **ARC AXS10x Projects**, **ARC EM Starter Kit Projects**, **ARC EM SDP Projects**, **ARC HS Development Kit Projects** and **ARC IoT Development Kit Projects**.

Note

Note: for each of these project types there is a list of toolchains which are supported by it.

- **ARC Cross ELF32 Target Application** and **ARC Cross ELF32 Target Static Library** support all of the toolchains;
- **ARC AXS101 Projects** supports ARC 600, ARC 700 and ARC EM toolchains;
- **ARC AXS103 Projects** support only ARC HS toolchain;
- **ARC EM Starter Kit Projects** – only ARC EM toolchain;
- **ARC EM SDP Projects** – only ARC EM toolchain;
- **ARC IoT Development Kit Projects** – only ARC EM toolchain;
- **ARC HS Development Kit Projects** – only ARC HS toolchain;

Project types are only available in the project creation dialog if at least one of the corresponding toolchain compilers is found in the *PATH* environment variable or in *../bin/* directory relative to Eclipse executable. Then you choose a project template, list of available toolchains appears on the right side of the dialog. There are only toolchains that are supported by this type of project and also found in the *PATH* or *../bin/* directory.

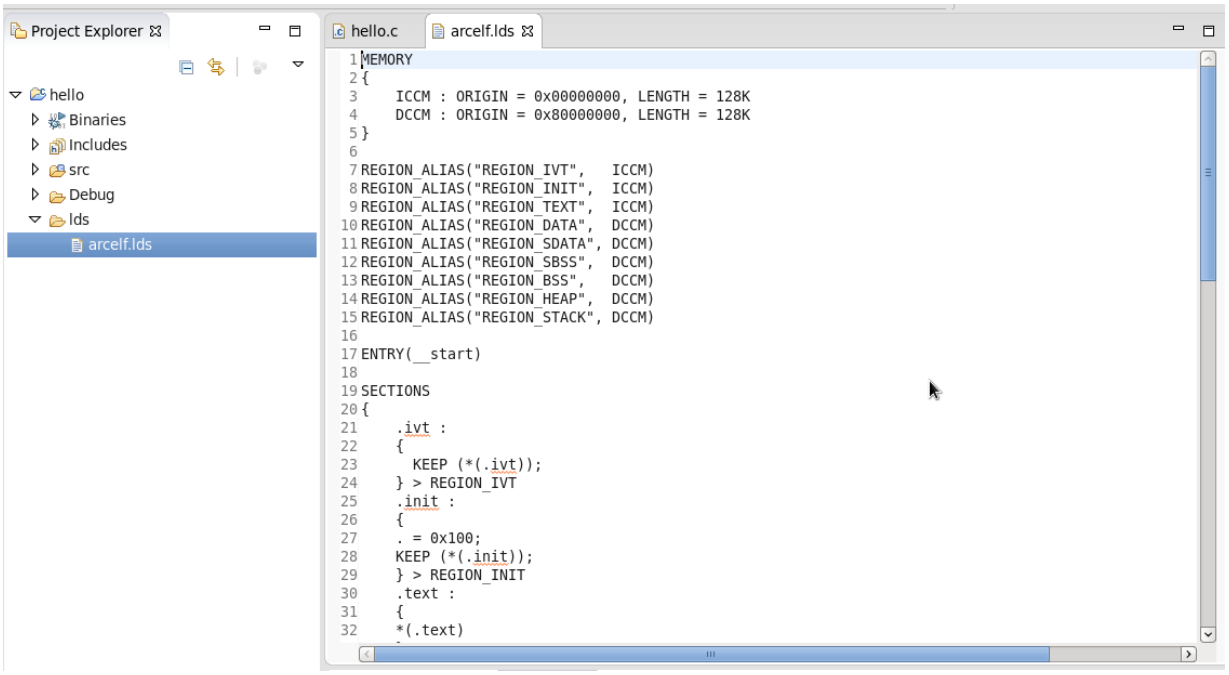


List of available toolchains for a template

If you want to create an application for nSIM, choose **ARC Cross ELF32 Target Application**. There you can choose either an empty or “Hello World” project template. Please note that this “Hello World” project calls `printf()` function, so it can not be used on hardware development systems, since they use UART for input/output and libc library does not provide bindings between UART and C standard library I/O functions. For nSIM this project will work fine, but for hardware development systems please choose “Hello World” projects that use UART. On the contrary, “Hello World” projects under **EM Starter Kit Projects** project type use UART, so they are not suitable for nSIM.

If you want to create a project for the Synopsys development system, choose one of projects that correspond to the system, for example **ARC EM Starter Kit Projects**. For each of these project types there is an **Empty Project** in the

list of templates and also **<board name> Empty Project** templates. If you want to create an empty project for your board, choose an empty project template that is specific for your board and core you are using. These templates contain memory maps of the cores, which are then passed to the linker. As for **Empty Project** templates, they are generated automatically by Eclipse and do not contain any specific information, so you would have to provide a memory map yourself, or your application might not work properly.



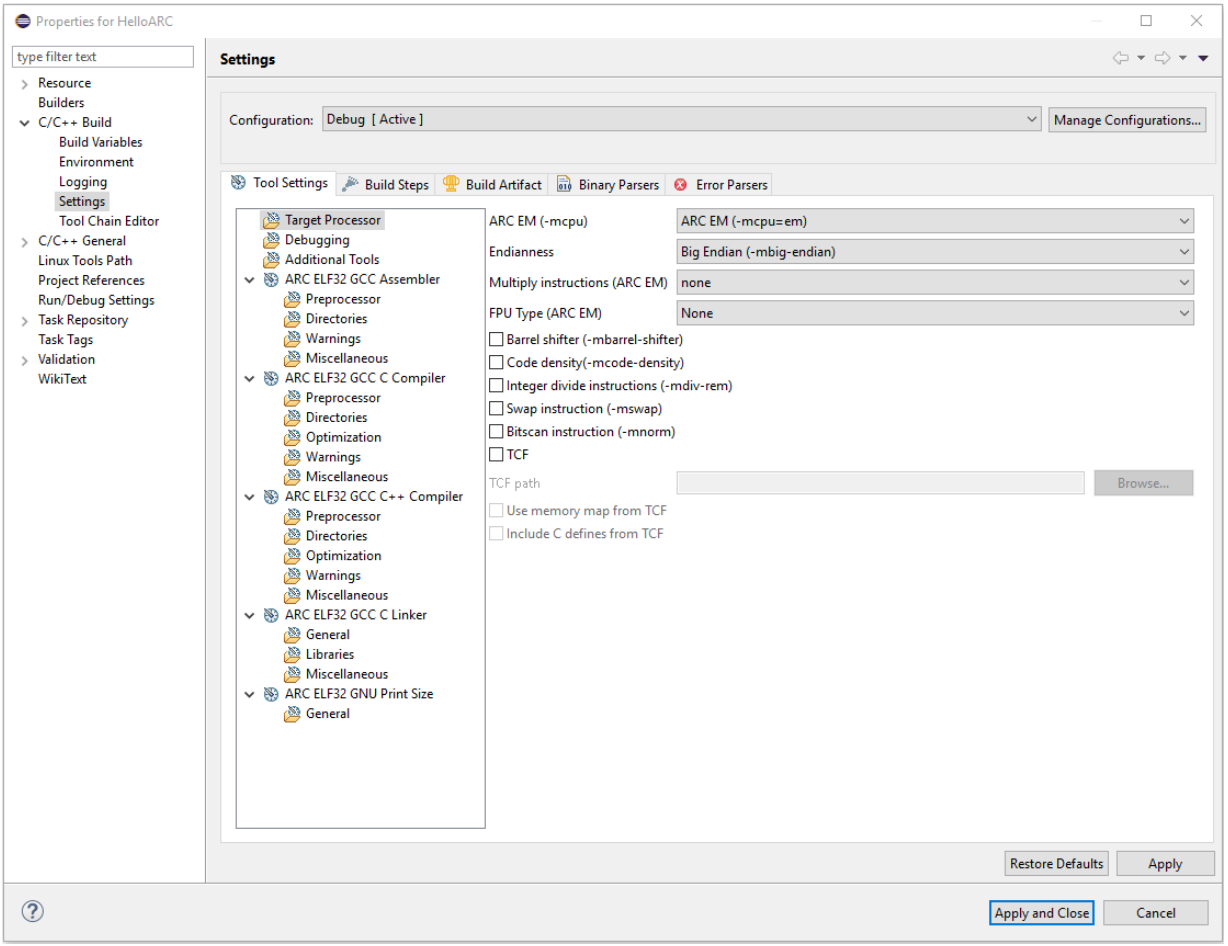
*Memory map for **Hello World for EM SK 2.1 Project***

> **Note**
>
> There is an ARC EM SDP project template which uses configuration for emsdp_em11d_dfss FPGA image. Projects created with this template may not work with other FPGA images of ARC EM SDP.

Building User Guide

Project Build Settings

To see your project build settings, right click on your project and select **Properties** from the pop-up menu. In the appeared dialog choose **C/C++ Build > Settings**, then select **Tool Settings** tab.

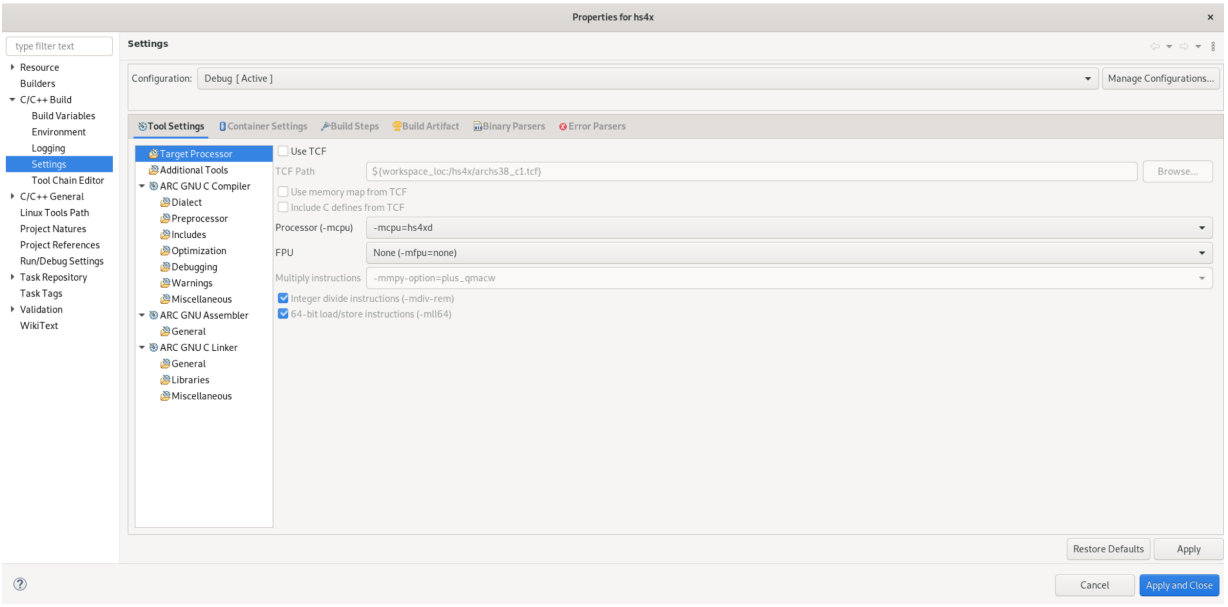


Project Build Settings page

At the left of the tab there is a list of tools which are used to build your project and **Target Processor**, **Debugging** and **Additional Tools** pages. For each of the listed tools there are some pages where you can set properties for these tools.

Target Processor Page

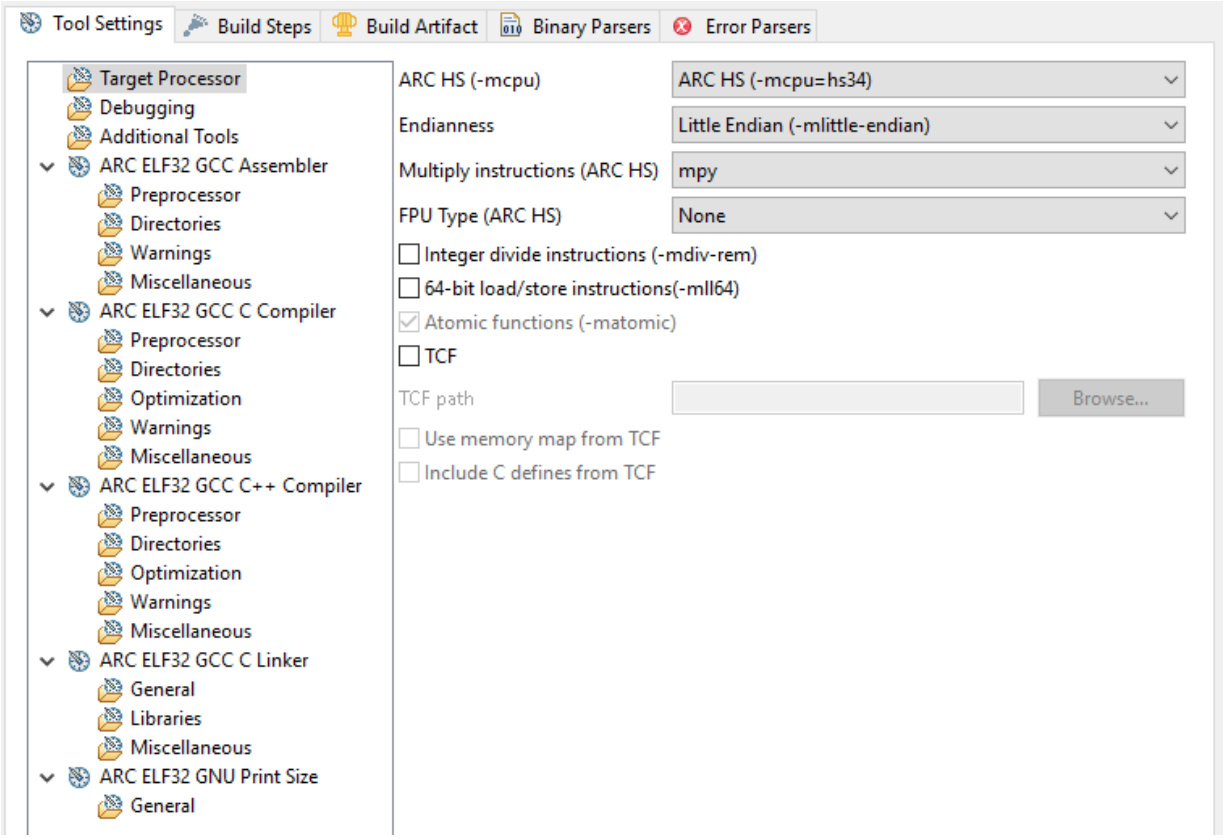
On this page there are properties that describe your target. These properties are different for different processors.



Target Processor Page for ARC HS

CPU option

CPU option for ARC 600 and ARC 700 has only one value, but for ARC EM there are several possible values: `arcem`, `em`, `em4`, `em4_dmips`, `em4_fpus` and `em4_fpuda`. Possible values for ARC HS CPU are `archs`, `hs`, `hs34`, `hs38`, `hs38_linux`, `hs4x` and `hs4xd`. For each of these values there are precompiled standard libraries that use some other target options. For example, if you choose `hs34` as you CPU, standard library that uses atomic functions and multiply option “mpy” will be used. Values of these options are set in IDE when CPU is selected and can not be changed to weaker values without changing CPU. For example, if DP FPU is selected as a result of selecting CPU, you can not change it to any of SP FPU values or “None”, but you can change it to DP FPU with extensions.



ARC HS Target Processor Page with `hs34` *selected as CPU value*

Here are the options that are required for each of CPU values:

CPU	Multiply	FPU	Barrel shifter	Code density	Integer divide	Bitscan	Swap
arcem	wlh1	none	+	+	-	-	-
em	none	none	-	-	-	-	-
em4	none	none	-	+	-	-	-
em4_dmips	wlh1	none	+	+	+	+	+
em4_fpus	wlh1	SP FPU	+	+	+	+	+
em4_fpuda	wlh1	FPU with double assist	+	+	+	+	+

CPU	Multiply	FPU	Integer divide	64-bit load/store	Atomic
archs	mpy	none	+	+	+
hs	none	none	-	-	-
hs34	mpy	none	-	-	+

CPU	Multiply	FPU	Integer divide	64-bit load/store	Atomic
hs38	plus_qmacw	none	+	+	+
hs38_linux	plus_qmacw	DP FPU with all extensions	+	+	+
hs4x	plus_qmacw	none	+	+	+
hs4xd	plus_qmacw	none	+	+	+

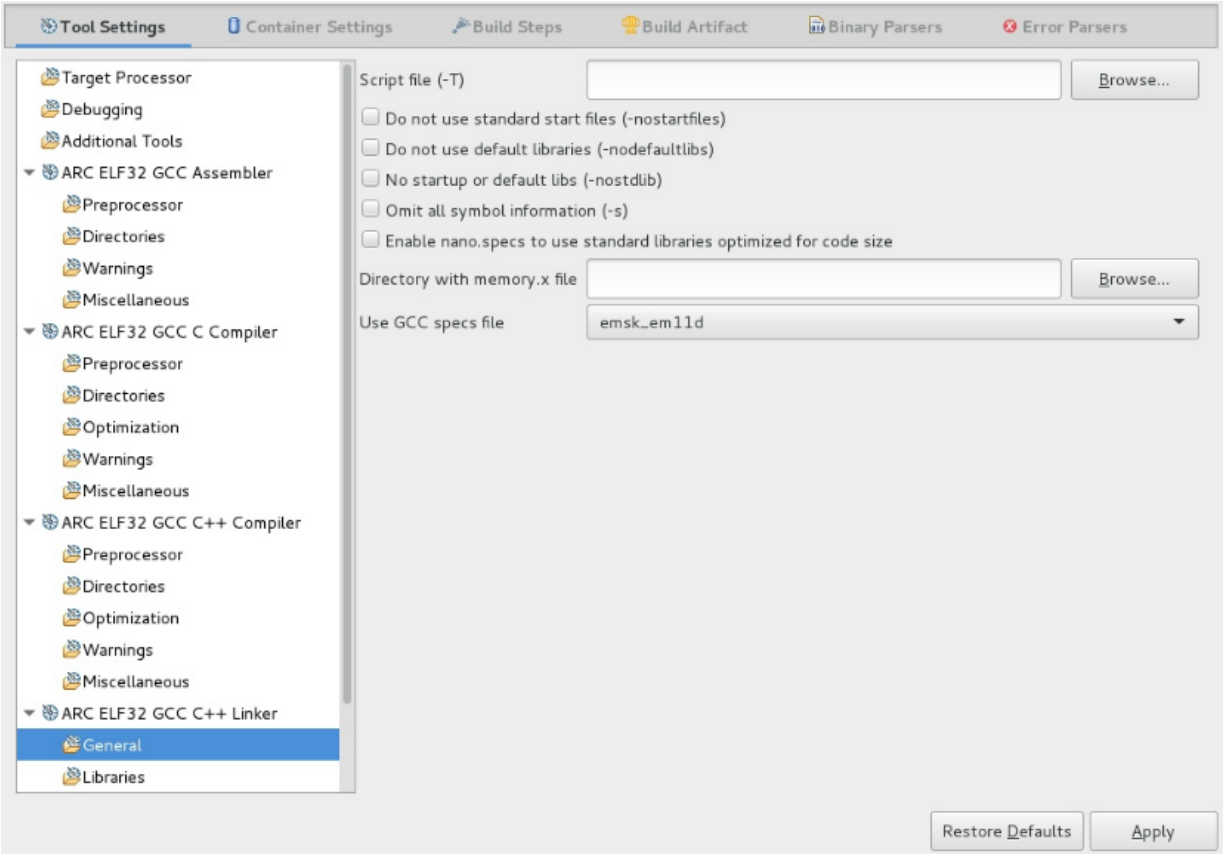
Note

Note that if you use TCF to provide target options, there are no checks that option values are consistent with CPU and you can specify there values that are weaker than CPU value requires. So please be careful when editing TCFs.

Linker options

Dropdown box in **ARC ELF32 GCC C/C++ Linker** > **General** allows to select which spec-file to use.

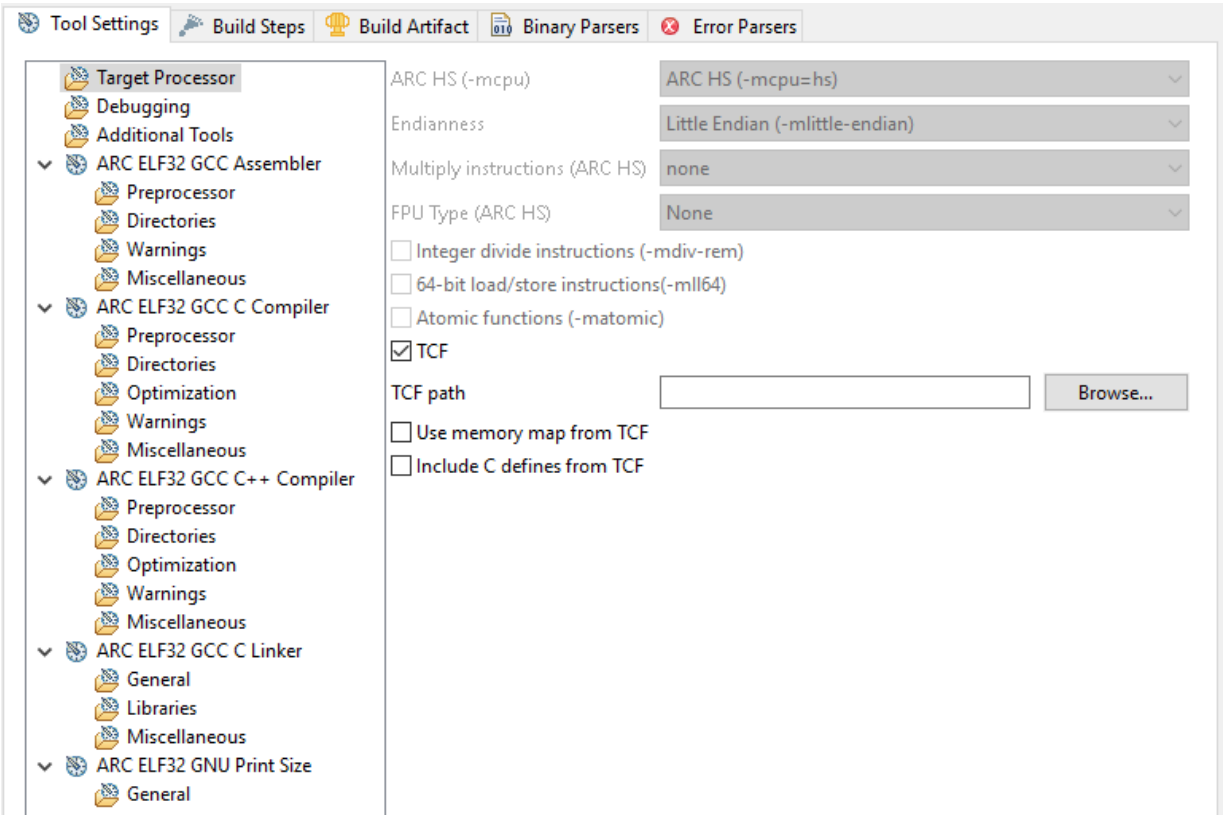
- **nsim** (`--specs=nsim.specs`)
- **nosys** (`--specs=nosys.specs`)
- **emsk_em9d** (`--specs=emsk_em9d.specs`)
- **emsk_em11d** (`--specs=emsk_em11d.specs`)
- **None** (don’t pass any option).



Dropdown list to select GCC specs file

Other options and TCF

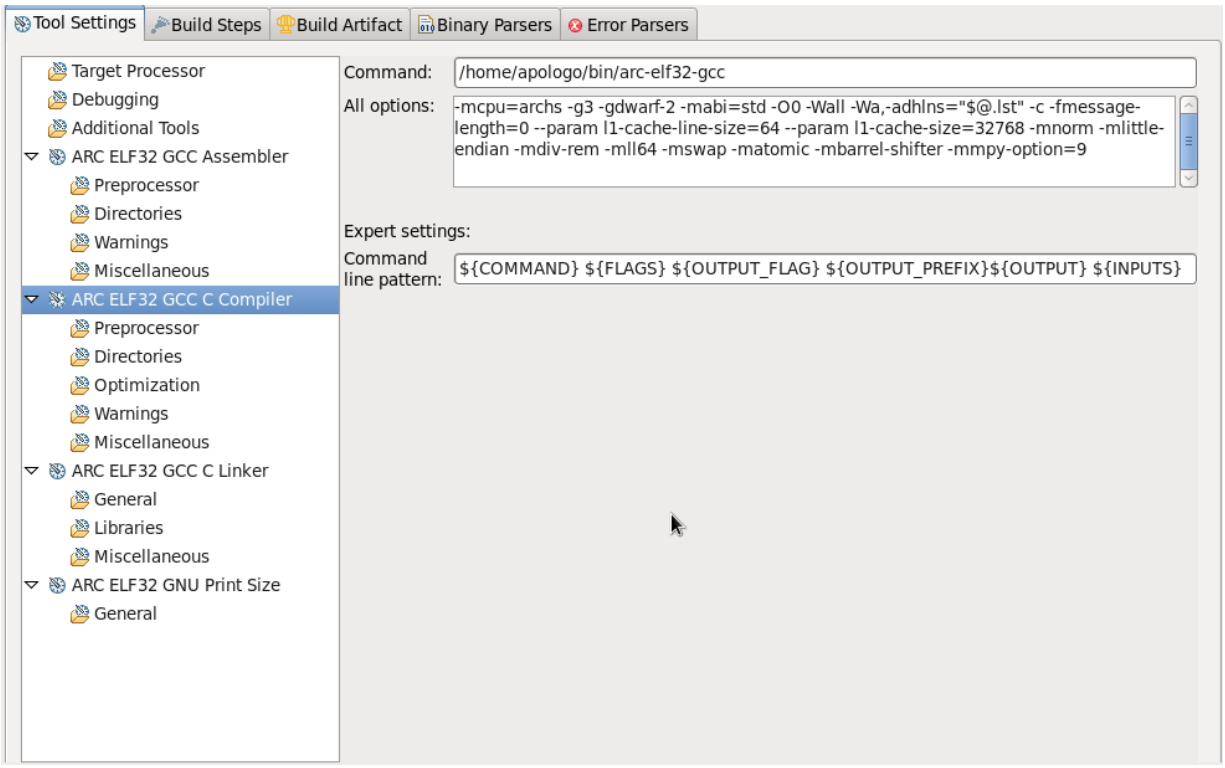
- Endianness is set when you choose a toolchain for your project and can not be changed.
- Other architecture options you can either set manually or choose a TCF file for used CPU core (available only for ARC EM and HS), which will set these options automatically. The only option that is not set automatically by selecting a TCF file is **ABI selection** option, which is available only for ARC HS processors.



Target Processor Page for ARC HS with TCF checkbox selected

It is recommended to use TCF files, because they are generated from the Build Configuration Registers and thus most reliably describe target core. TCF files are provided by your chip designer.

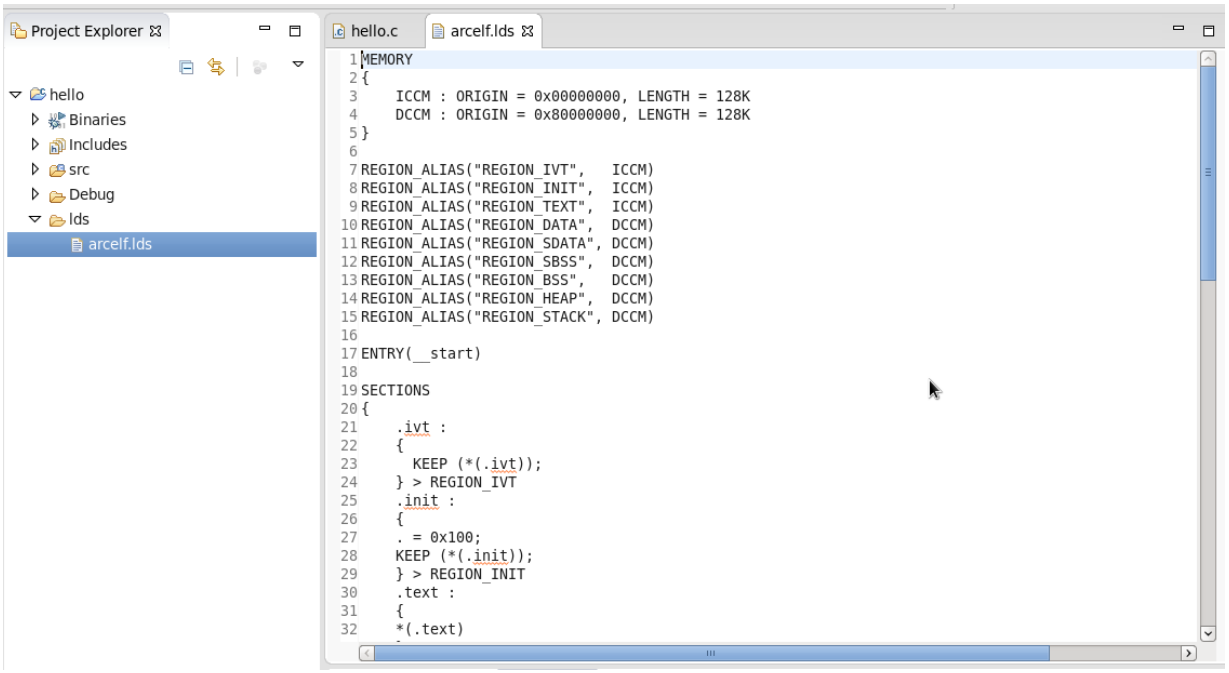
To see which options are set automatically if TCF is chosen, you can select a tool from the list on the left of the dialog and see the list of options to be passed to this tool in the **All options** field.



List of all the options to be passed to compilerx

If **TCF** is selected, **Use memory map from TCF** and **Include C defines from TCF** checkboxes become enabled. If you check **Use memory map from TCF** box, memory map from TCF file will be passed to the linker. Checking **Include C defines from TCF** includes C defines from TCF.

Note that templates from **AXS10x Projects** and **EM Starter Kit Projects** already contain memory maps that are used if no other is provided. However, this is true only for **Hello World for EM SK** and **Empty Project For** templates, but not **Empty Project** ones. **Empty Project** templates are generated automatically by Eclipse and do not contain any specific information.



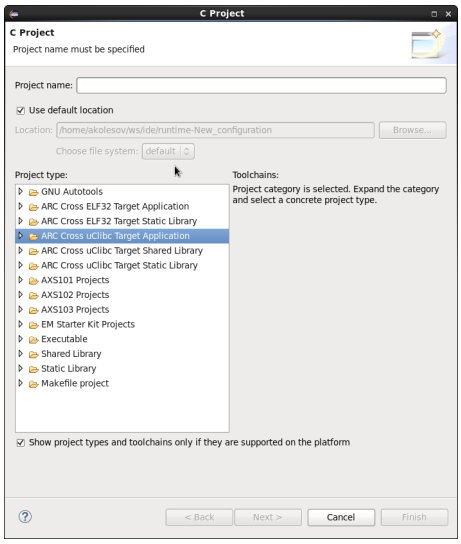
*Memory map for **Hello World for EM SK 2.1 Project***

Building Linux uClibc applications

The C Project dialog has five ARC project types on Linux: ARC Cross ELF32 Target Application, ARC Cross ELF32 Target Static Library, ARC Cross uClibc Target Application, ARC Cross uClibc Target Shared Library and ARC Cross uClibc Target Static Library.

Creating a New C Project

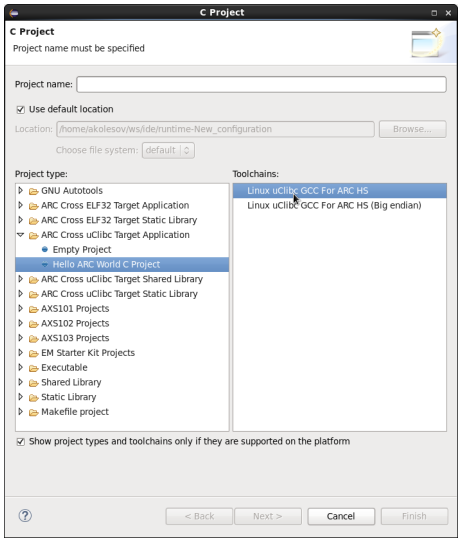
1. Select File >New >C Project



C Project Types on Linux

Choosing toolchain

1. Choose proper toolchain for uClibc project type.



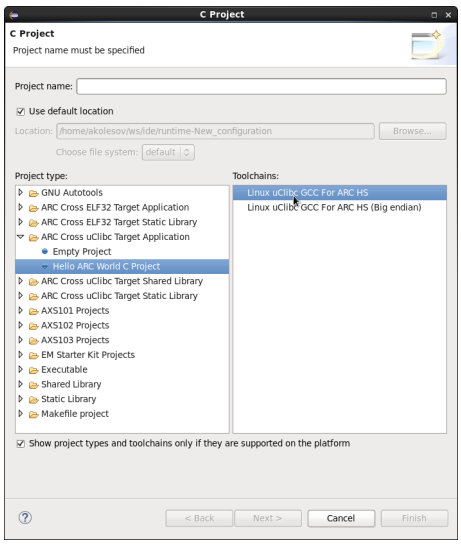
uClibc supported toolchain

Setting compile options based on CPU core

User should choose a proper toolchain for a core, for different core supports different compile options.

Compiling a uClibc application

1. Select File >New >C Project



Hello_uClibc_Application

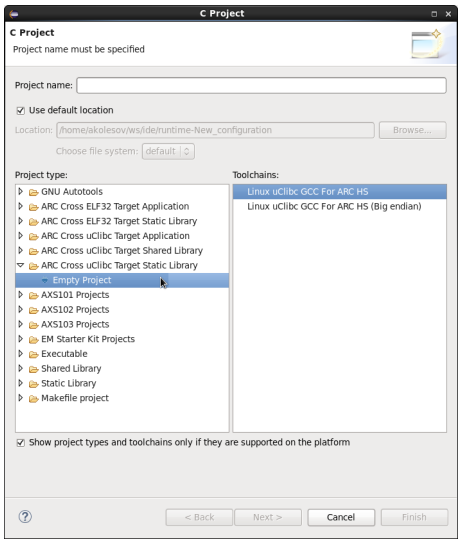
2. Getting compiling output in console

Creating and building an application

Hello_uClibc_Application_700_output

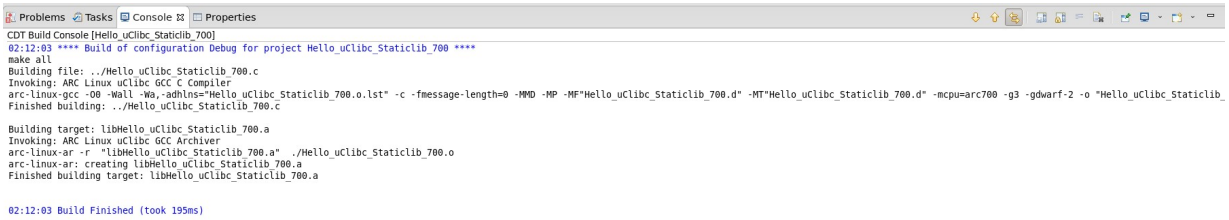
Compiling a uClibc Static Library

1. Select File >New >C Project



Hello_uClibc_Staticlib

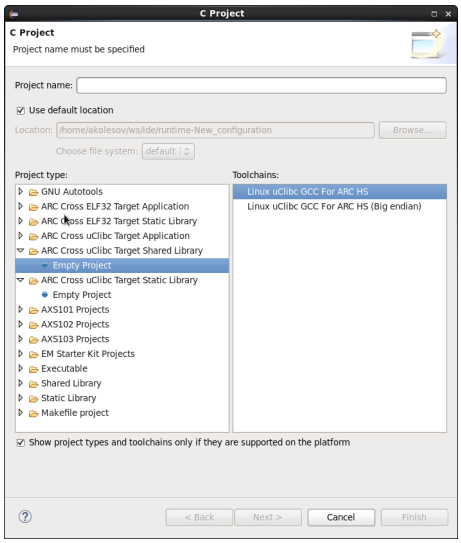
2. Getting compiling output in console



Hello_uClibc_Staticlib_700_output

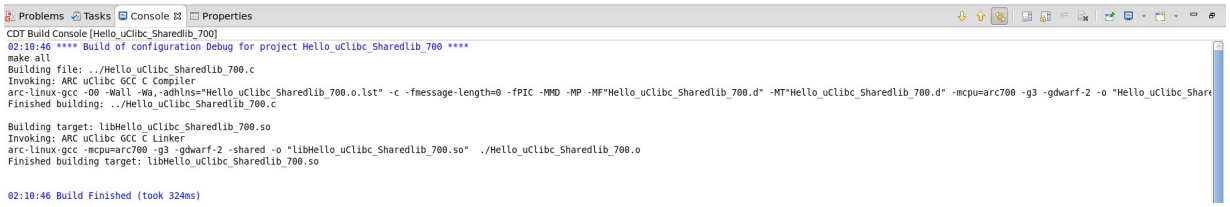
Compiling a uClibc Shared Library

1. Select File >New >C Project



Hello_uClibc_Sharedlib

2. Getting compiling output in console



Hello_uClibc_Sharedlib_700_output

How to Use Custom Toolchain

You might want to use an external toolchain (for example, built for a particular CPU configuration) instead of the one shipped with the IDE installer. Currently there is only one way this can be done: external toolchain location should be added to the beginning of the `PATH` environment variable.

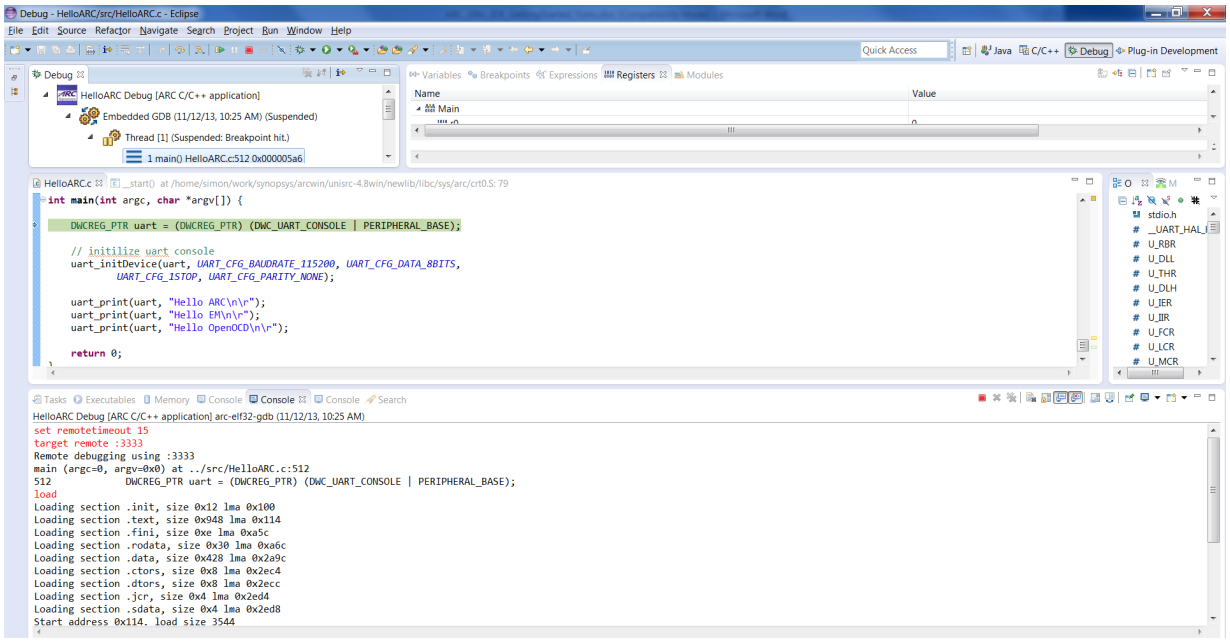
To create a project using external toolchain added to `PATH`, open C project creation dialog and select one of ARC project types. For the list of available project types and toolchains supported by them, see ARC Project Templates. Note that project should be created with a target toolchain already in the PATH, otherwise it will use standard library headers from the original toolchain with which it was created.

As it is explained on ARC Project Templates page, IDE allows you to create projects only if supported toolchains compiler is found in `PATH` or in `../bin/` directory relative to Eclipse executable, so if there are other toolchains present there except your external toolchain, projects that support them will be available too. However your external toolchain will hide other toolchains present in `PATH` or `../bin/` that contain the same tools as yours, so you will not be able to create projects that use them.

Debugging

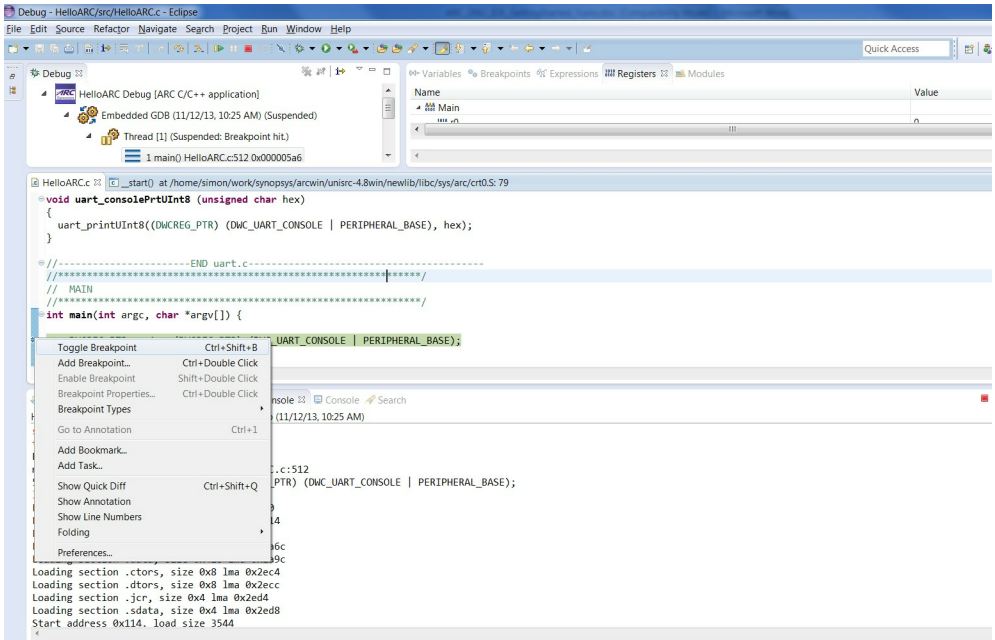
Using Debug Perspective

The **Debug** perspective provides an integrated debug environment with individual windows to display various debugging data such as the debug stack, variables, registers breakpoints, etc.



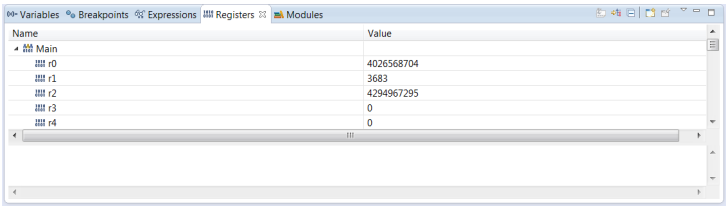
Debug Perspective

1. To set a breakpoint, place your cursor on the marker bar along the left edge of the editor window on the line where you want the breakpoint:



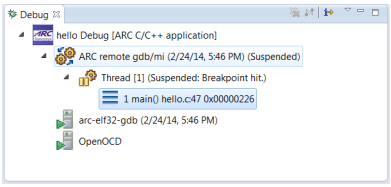
Source File Window in Debug Perspective with Breakpoint Set

2. Examine Variables, Breakpoints, Expressions or Registers from different tabs of the same debug perspective:

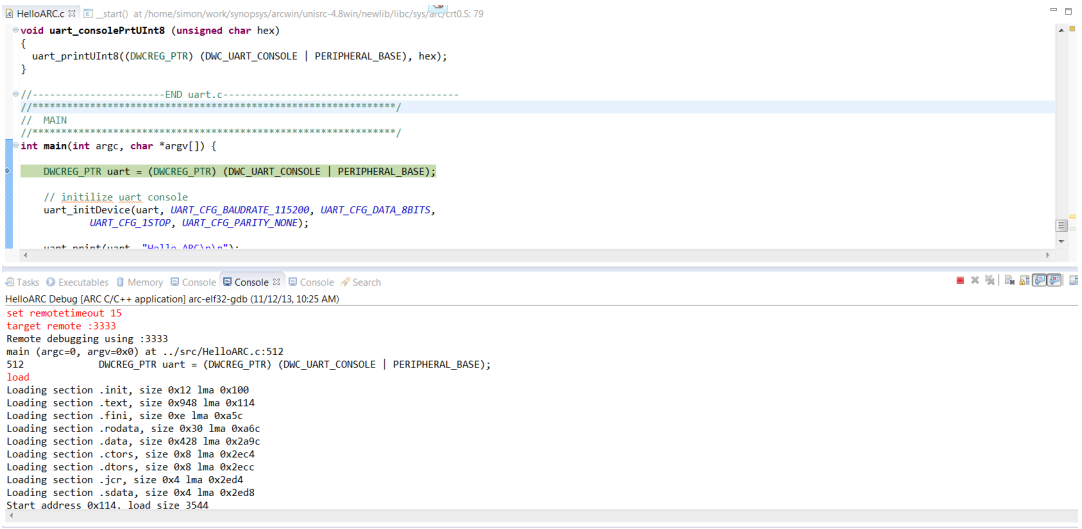


Registers Window in Debug Perspective

3. Examine the debug Views showing the debugger in use:

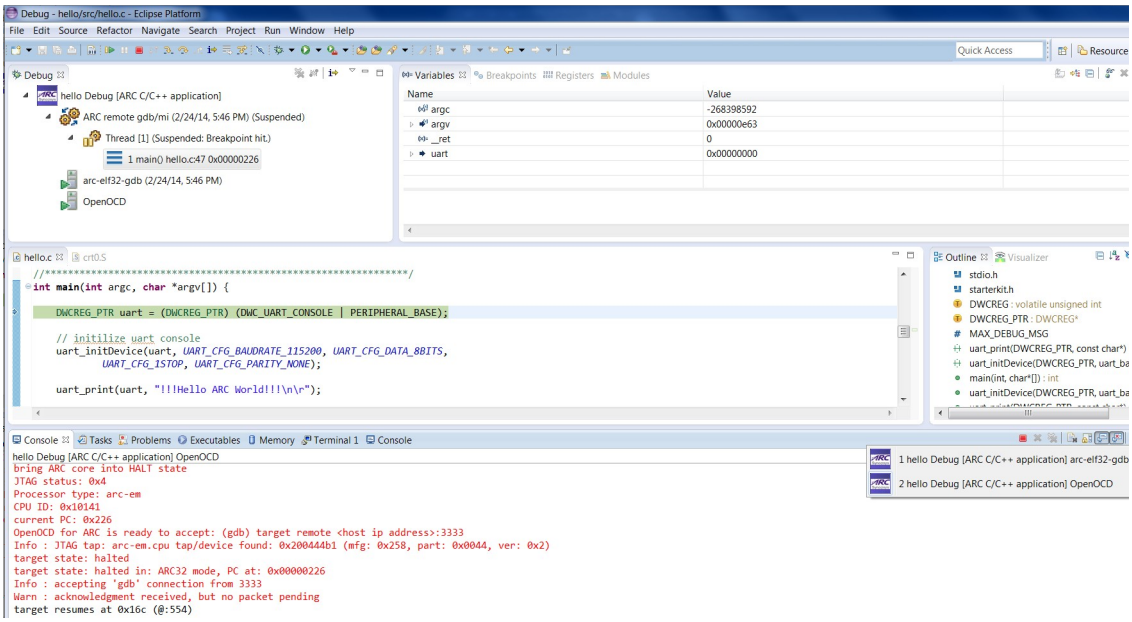


Debug Window in Debug Perspective

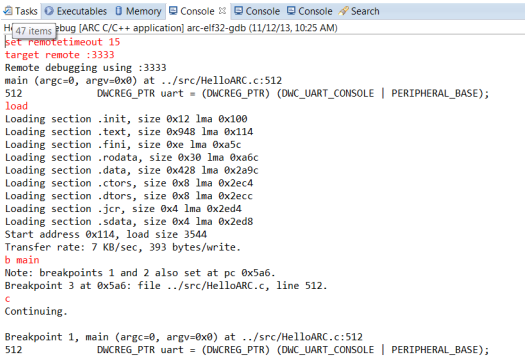


Hello ARC Debug Console in Debug Perspective

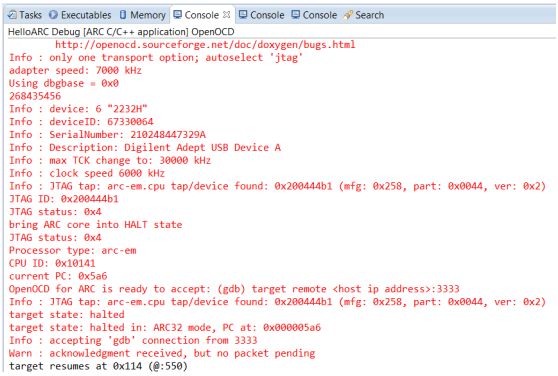
4. Switch Console tabs to view OpenOCD **Console** output:



Multiple Consoles in the Debug Perspective



Hello ARC Debug Console Output

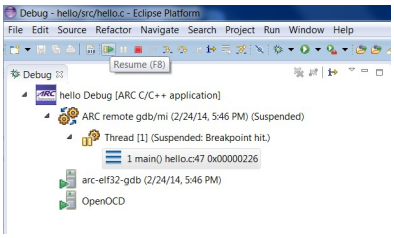


OpenOCD Console Output

5. Step through each line by using F5 (step into), and F6 (step over).

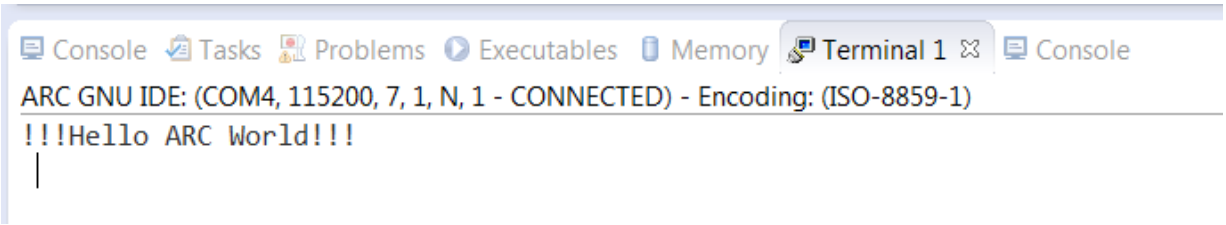
Stepping Toolbar

6. Toggle breakpoint at the last line of main(), which is “}” , and then click Resume or press F8.



Click Resume or Press F8

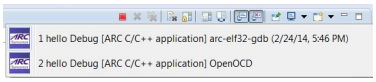
7. To see the UART output, open Eclipse Terminal view.



Final Output Printed to Serial Terminal Window through UART

You will be able to see the output in the Terminal view only if COM port specified in **Terminal** tab of **Debug Configurations** dialog is right. Read more about specifying a COM port Setting a COM port

8. Terminate all external tools before you quit current debugging process.



Consoles for child processes

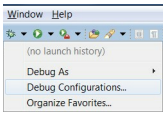
Creating a Debug Configuration

Creating a new debug configuration

Once the C Project is successfully compiled by ARC GCC, you can debug the resulting executable on a board or using nSIM.

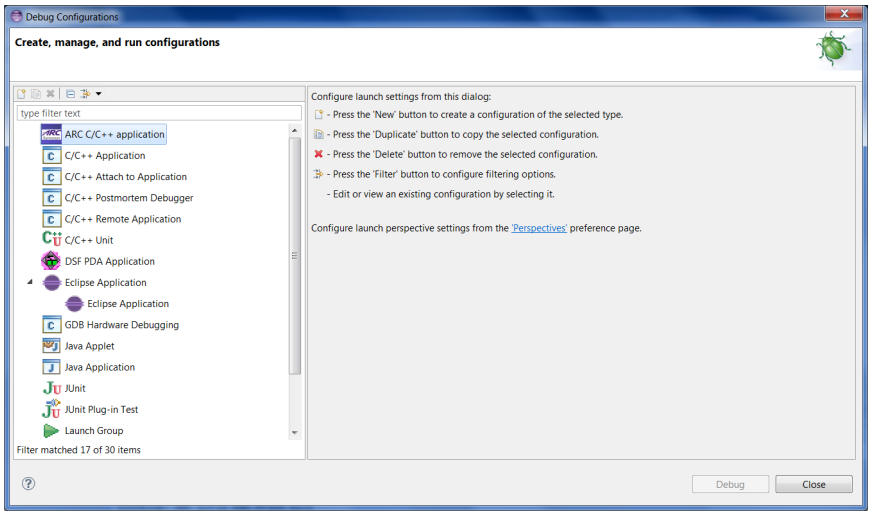
To debug the project, create a new debug configuration.

1. Select **Debug Configurations** from the **Run** menu or by clicking on the down arrow next to the bug icon:



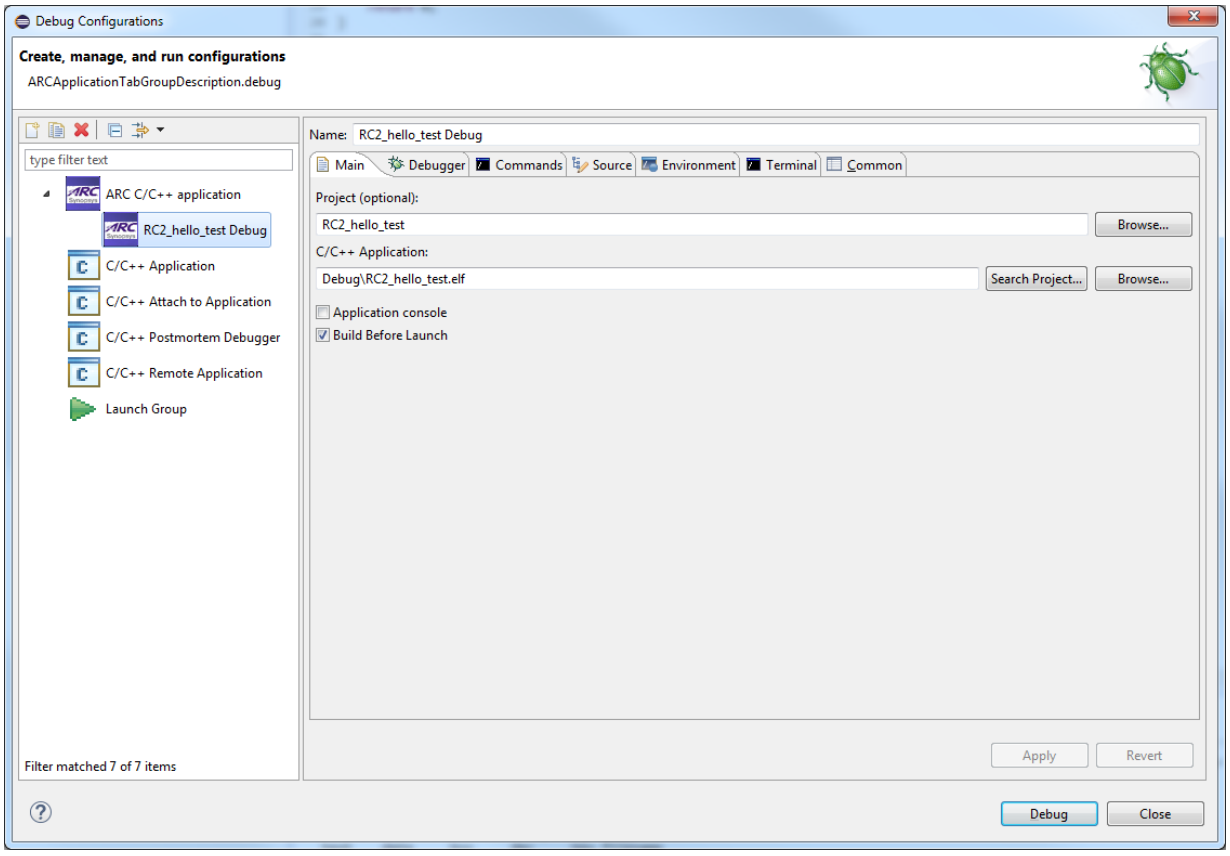
Debug Configurations

2. Double click on the **ARC C/C++ Application** or click on the top left icon to create a new debug configuration for the project:



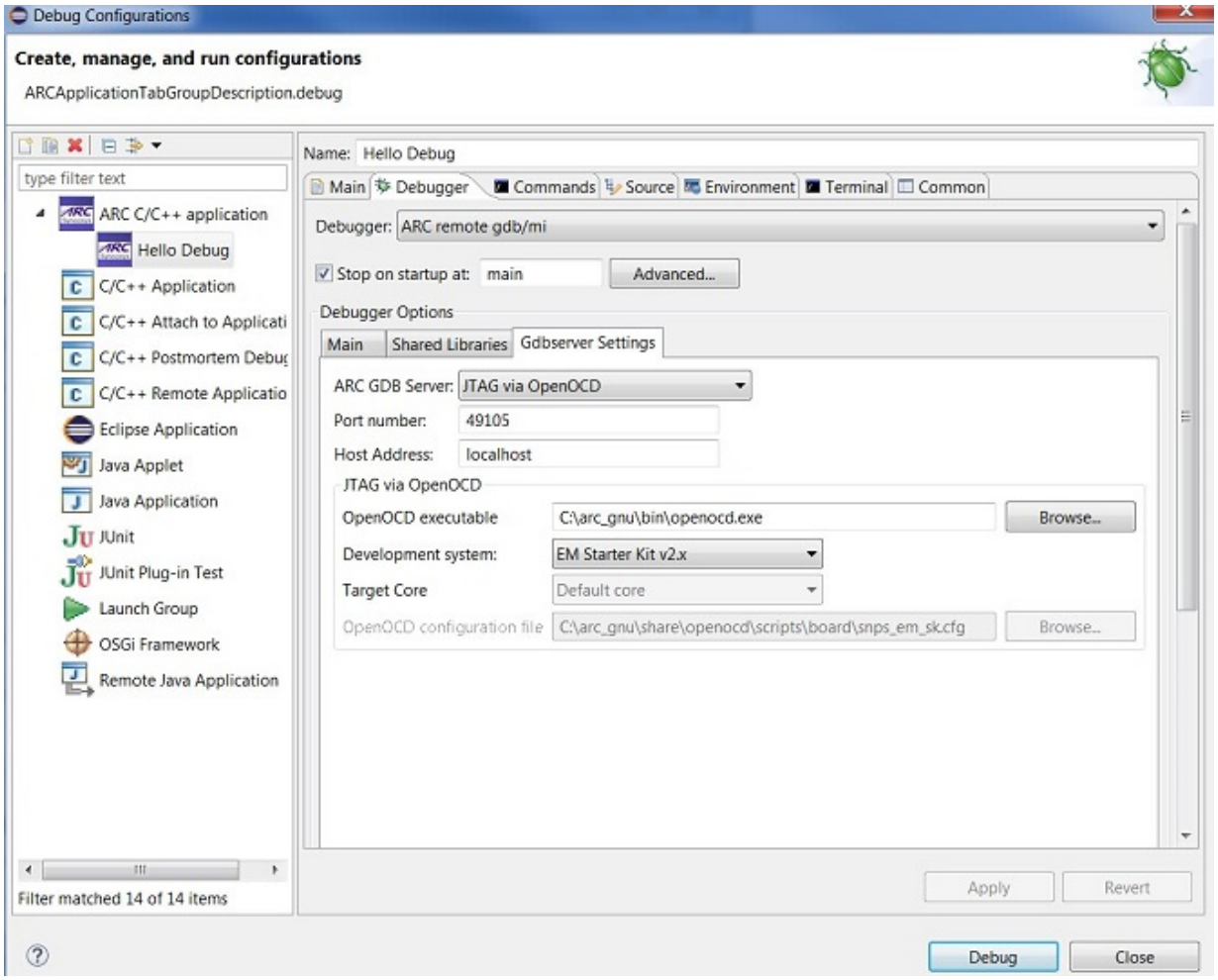
ARC Embedded Debug Configurations

3. Select a name for the new debug configuration (by default, it equals the project name followed by “Debug”).



New debug Configuration

4. Click the **Debugger** tab.



Default values in the Debugger tab for JTAG via OpenOCD

Here you can select a GDB server you want to use. About different GDB servers and their settings see pages

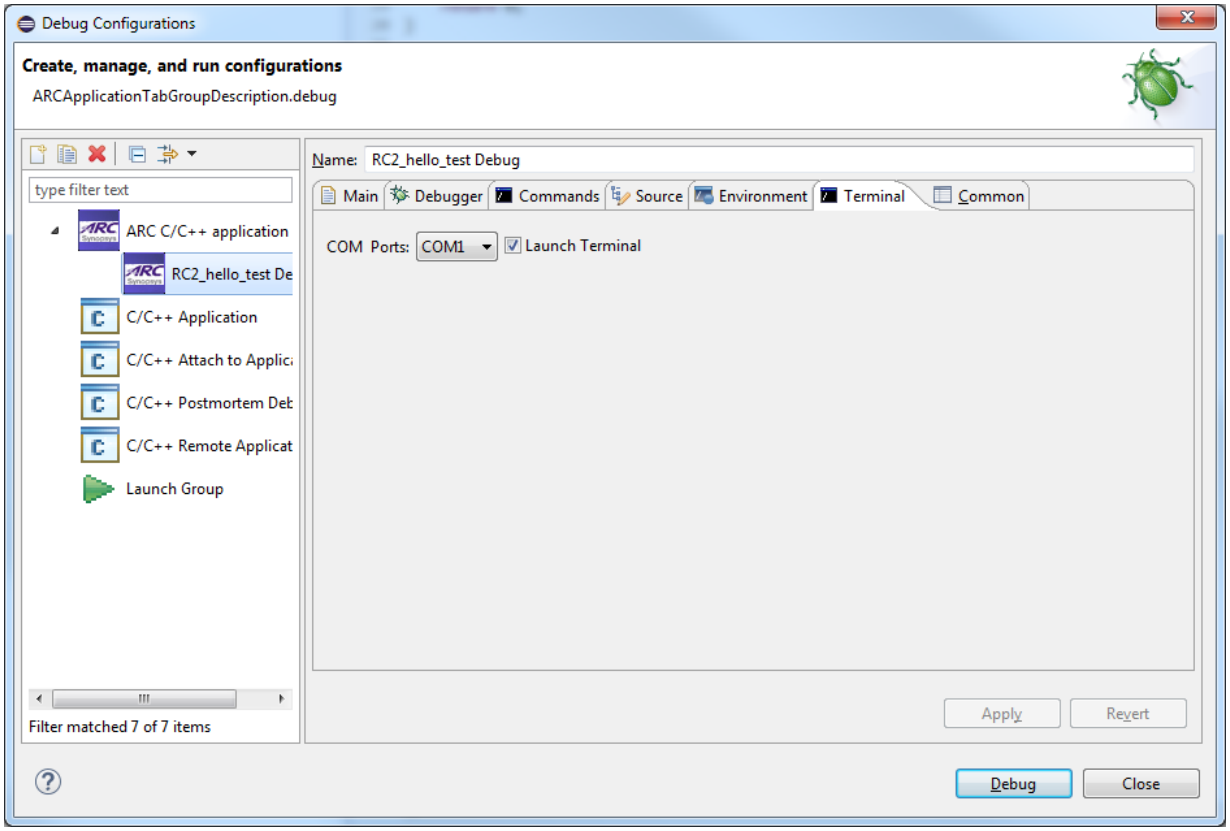
- *Debugging with OpenOCD <debugging-with-openocd>*
- *Debugging with Opella-XD <debugging-with-opellaxd>*
- *Debugging with nSIM <debugging-with-nsim>*
- *Debugging using custom GDB server <debugging-with-custom-gdb-server>*
- *Using running GDB server <debugging-with-running-gdb-server>*

Note

There is a known problem with changing **ARC GDB Server**'s value on Ubuntu. After changing the value there are only two fields visible: **ARC GDB Server** and **Port number**. Workaround: select GDB server's value, press **Apply** button, then close and open the dialog again. After that all the necessary fields become visible.

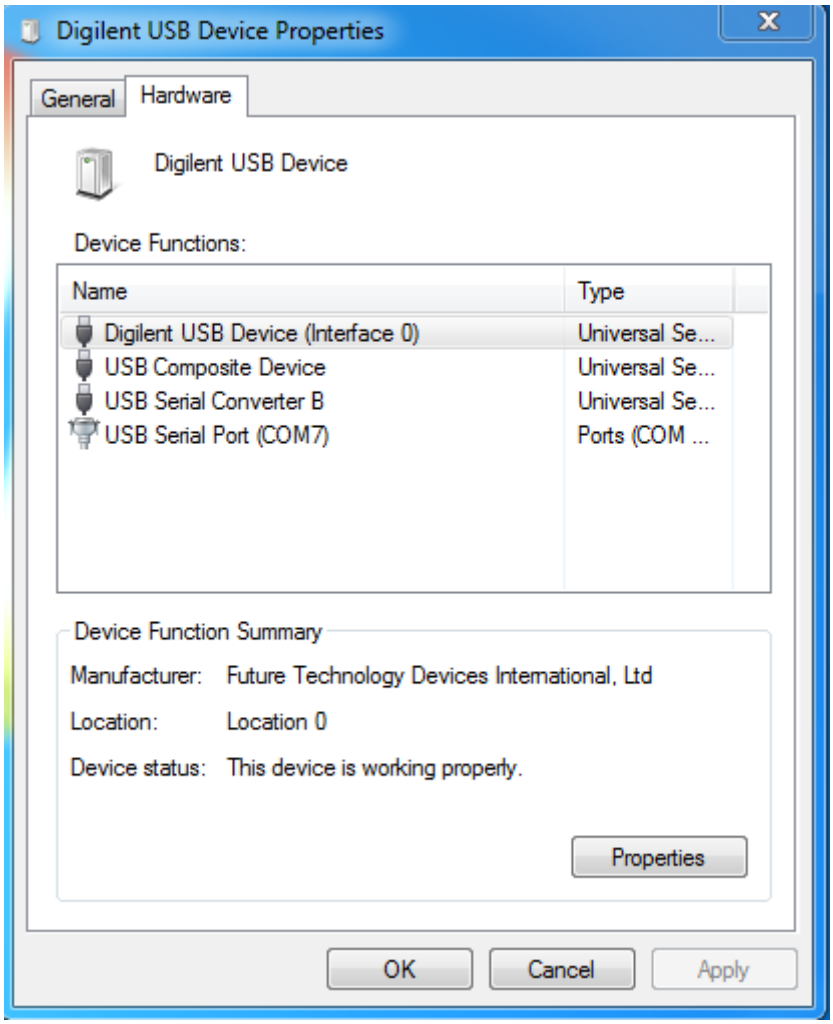
Setting a COM port

If you are debugging an application on a board you need to specify a COM port to connect to. Open the **Terminal** tab.



Terminal Tab

The **COM Ports** picklist shows the value for Digilent USB Serial Port from the Windows registry. You can modify the value as desired, but the selection must match the port number in Device and Printers as shown in below.



Digilent USB Serial COM Port

Starting a debug session

1. Click the **Debug** button in the **Debug configurations** dialog or **Debug** button of IDE to initiate debug session.

   This action automatically launches your GDB server (if you are not connecting to a running one). If you are using a board, it also launches the Serial terminal and connects to your board.

2. Click **Yes** in the confirmation dialog to switch to the Debug perspective.

Debugging with OpenOCD

It is expected here that you have already built your application and created a debug configuration for it. About how to do it you can read on the following pages:

- Building an Application
- Creating a Debug Configuration

Board Configuration

For AXS 10x board configuration refer to Board configuration and User Guide of AXC00x CPU Card you are using.

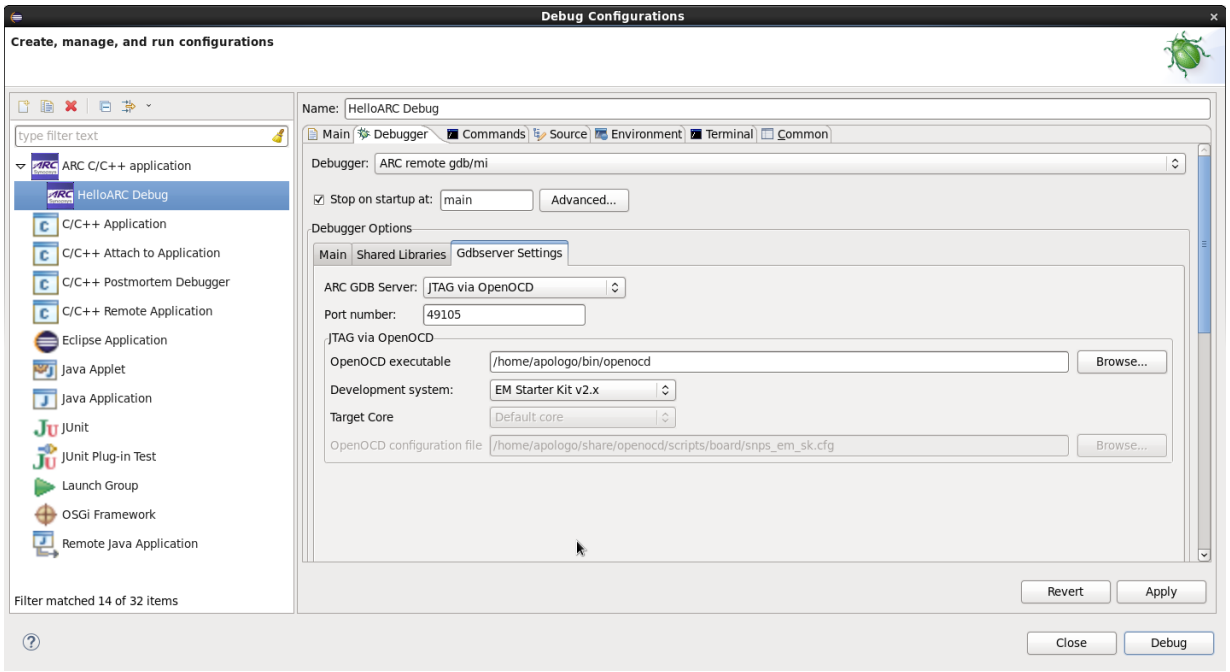
For EM Starter Kit use default configuration.

Configuring drivers on Windows

If you are using Windows, you should configure drivers for your device before you start.

About how to do it see How to Use OpenOCD on Windows.

Specifying OpenOCD properties



OpenOCD debugger tab

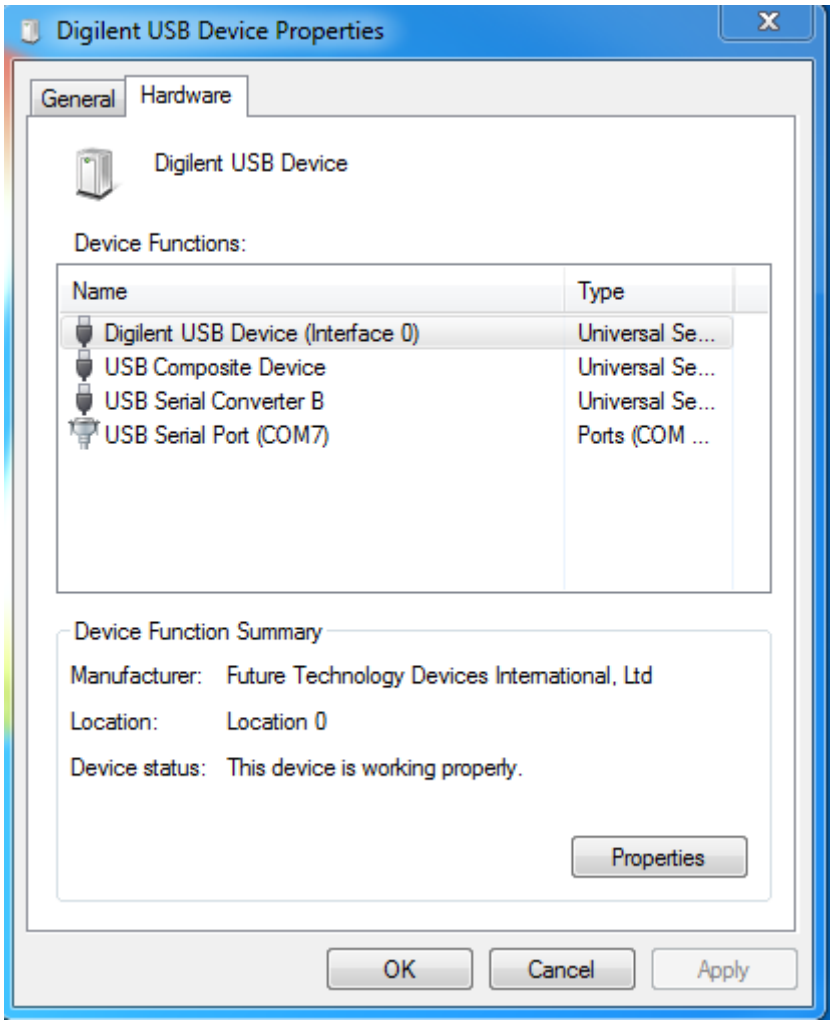
In this tab you can choose your development system and then in the **OpenOCD configuration file** field you will see a path to a file that will be used by OpenOCD. If you want to use another configuration file, you can choose **Custom configuration file** under **Development system** and select your own file in the enabled **OpenOCD configuration file** field.

Warning

The ARC GNU IDE Debugger plugin doesn’t support ARC EM SDP board in the GUI, however board configuration is compatible with the ARC EM Starter Kit 2.3 board. Therefore to debug application on ARC EM SDP, select ARC EM Starter Kit in the **Development system** menu.

Choosing COM Port

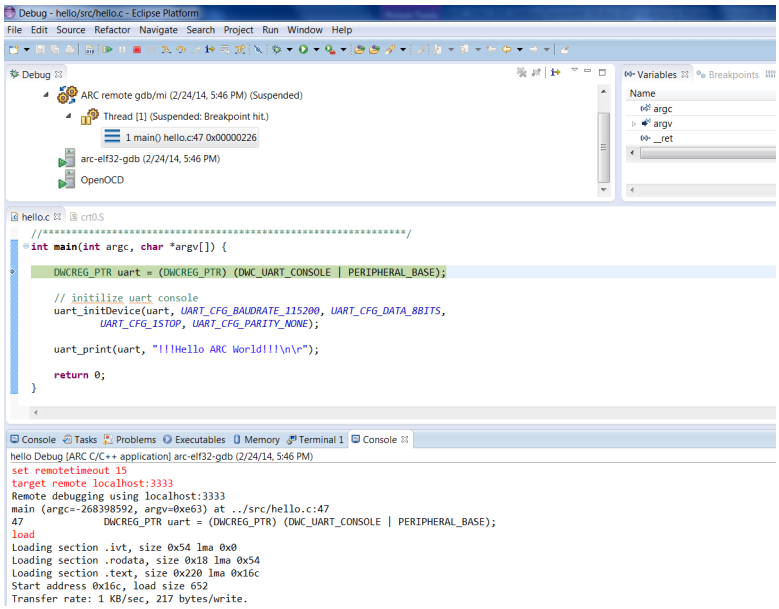
Open **Terminal** tab and select COM Port from the list. On Linux select **/dev/ttyUSB1** for EM Starter Kit and **/dev/ttyUSB0** for AXS10x. On Windows select COM port matching the port number from “Devices and Printers”:



USB Serial COM Port

Starting a debug session

To debug an application using OpenOCD, press **Debug** button of IDE and confirm switching to Debug Perspective.



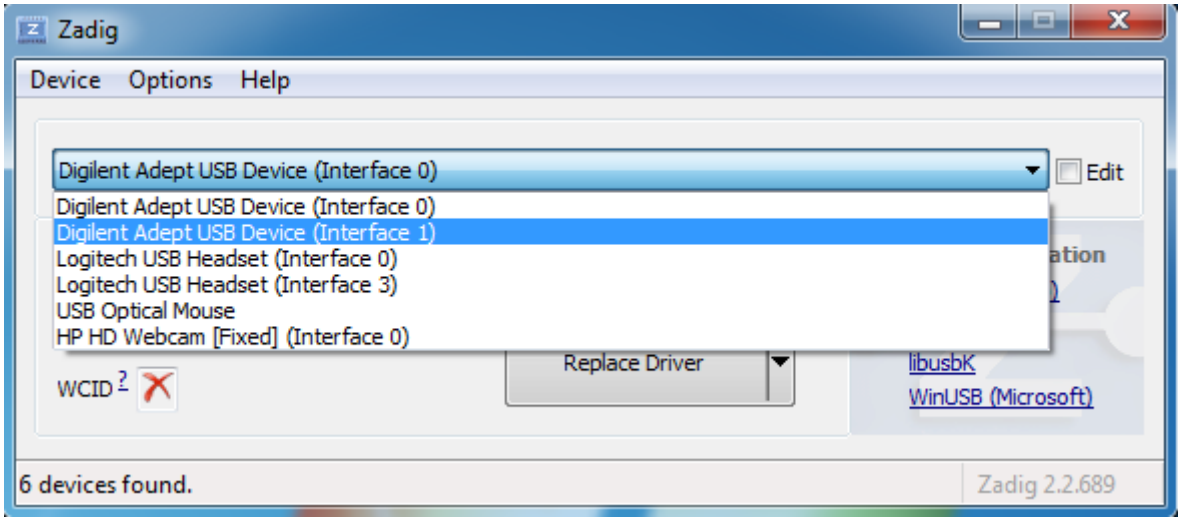
Debugging Process

How to Use OpenOCD on Windows

Replacing a driver

Before you can start using OpenOCD, you have to download WinUSB driver and replace with it one of FTDI drivers for your hardware development system.

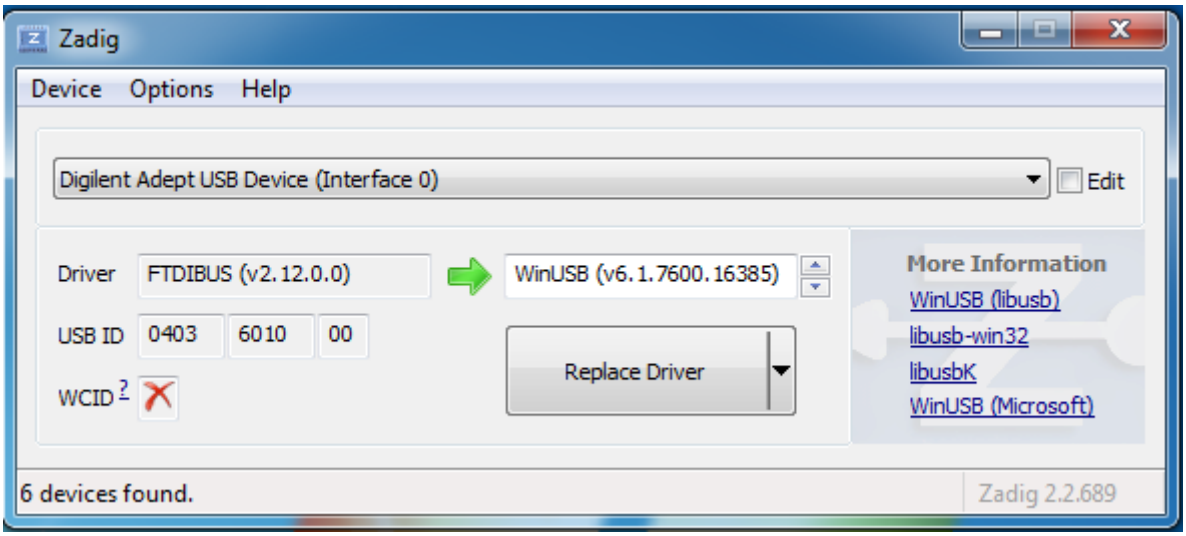
To do that, download Zadig and run it. You should be able to see **Digilent Adept USB Device** in the list of devices.



Device List

If your device is not shown by Zadig, tick **List all devices** in **Options**.

For EM Starter Kit, select **Digilent Adept USB Device (Interface 0)**, choose **WinUSB** driver and press **Replace Driver**. Your FTDI driver will be replaced with WinUSB.



Zadig Dialog

For ARC HS Development Kit, IoT Development Kit and AXS10x SDP, the only thing that differs is that instead of **Digilent Adept USB Device (Interface 0)** you should select **Digilent Adept USB Device (Interface 1)**.

Note that antivirus might complain about drivers files created by Zadig.

> **Note**
>
> If you want to change driver for your device back for some reason, you can uninstall current driver in "Devices and Printers" and then reconnect your board to the computer, Windows will install the default driver automatically.

Debugging with OpellaXD

It is expected here that you have already built your application and created a debug configuration for it. Please refer to the following pages for more information:

- Building an Application
- Creating a Debug Configuration

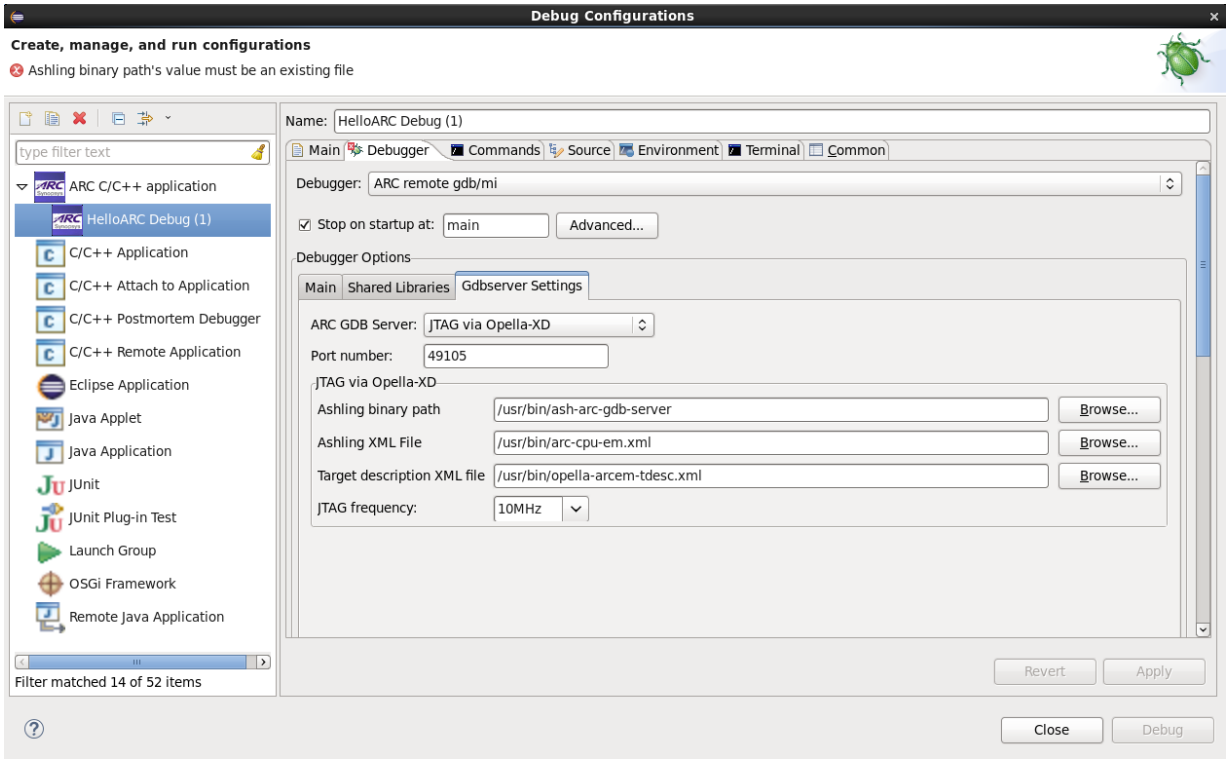
> **Note**
>
> Opella-XD has some problems, see section Known issues.
>
> On some platforms debugging fails when Opella-XD gdb server is started from the IDE, in this case use Alternative way of debugging.

Board Configuration

Board should be configured mostly the same way as for OpenOCD, see Board Configuration.

However, for AXS it is necessary to change some jumper settings when comparing to OpenOCD configuration. Please refer to AXS with Opella-XD board configuration.

Specifying properties for Opella-XD



Opella-XD on debugger tab

In this tab you should specify paths to your ashling executable file and two XML files. Both these files you can find here. In the **Ashling XML File** field you should choose one of `arc600-cpu.xml`, `arc700-cpu.xml`, `arc-em-cpu.xml` and `arc-hs-cpu.xml`. In the **Target description XML file** should be path to `opella-YOUR_CPU-tdesc.xml`. Note that file `aux-minimal.xml` should be also downloaded from that folder and put into the same folder as `opella-YOUR_CPU-tdesc.xml`. This file contains description common to all architectures and is included by all “tdesc” files.

JTAG frequency should be set to **7 MHz** for EM Starter Kit 2.0 and 2.1. For EM Starter Kit 2.2 select **5 MHZ**. For other hardware development systems leave **10 MHz**.

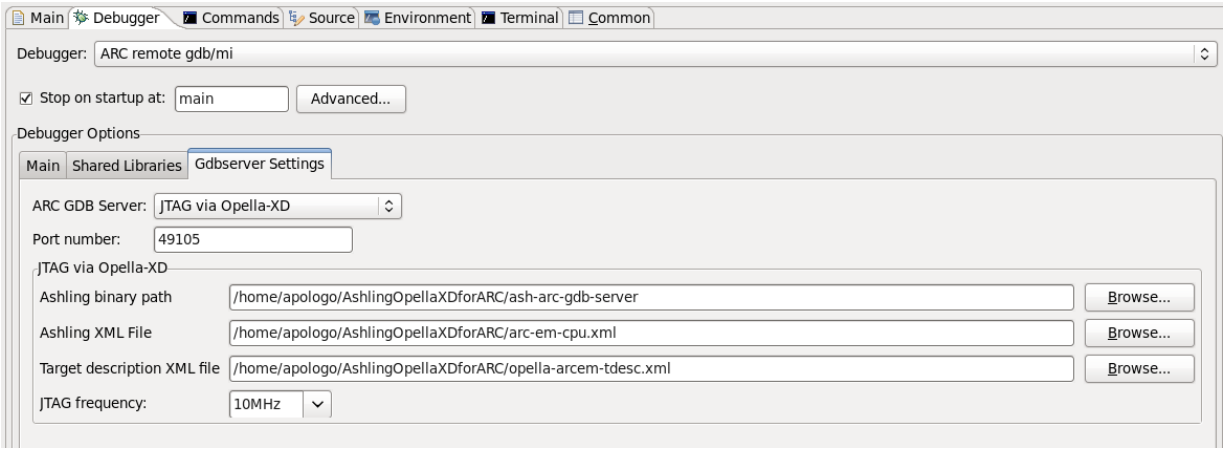
> **Note**
>
> Note that if you are using Opella-XD, you can not specify the core to debug, so you will be able to debug your application only if you have just one core in your JTAG chain.

Currently IDE always passes option `--device arc` to Opella-XD GDB-server which means that server would be configured to work with TPAOP-ARC20-R0 cable. Server configured in such way doesn’t work with TPAOP-ARC20-R1 - this cable requires `--device arc-jtag-tpa-r1` or `--device arc-cjtag-tpa-r1` option to be passed to Opella-XD GDB-server. As a consequence currently GNU IDE supports only TPAOP-ARC20-R0 cable.

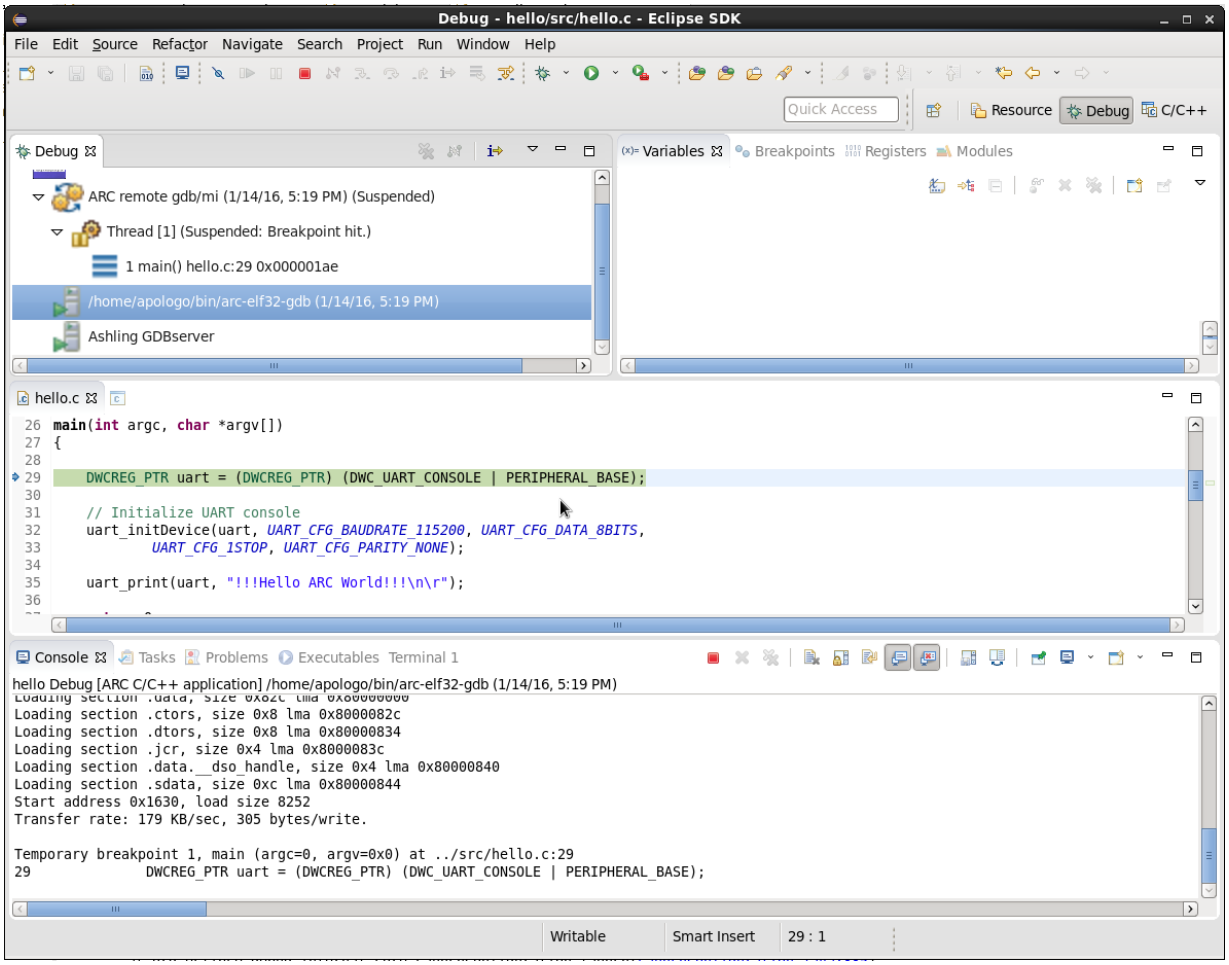
> **Warning**
>
> GDB on Windows can’t read XML files with Windows line endings (CR/LF) - tdesc XML file must be converted to UNIX line endings (LF).

Starting a debug session

To debug an application using Ashling Opella-XD, press **Debug** button of IDE and confirm switching to Debug Perspective.



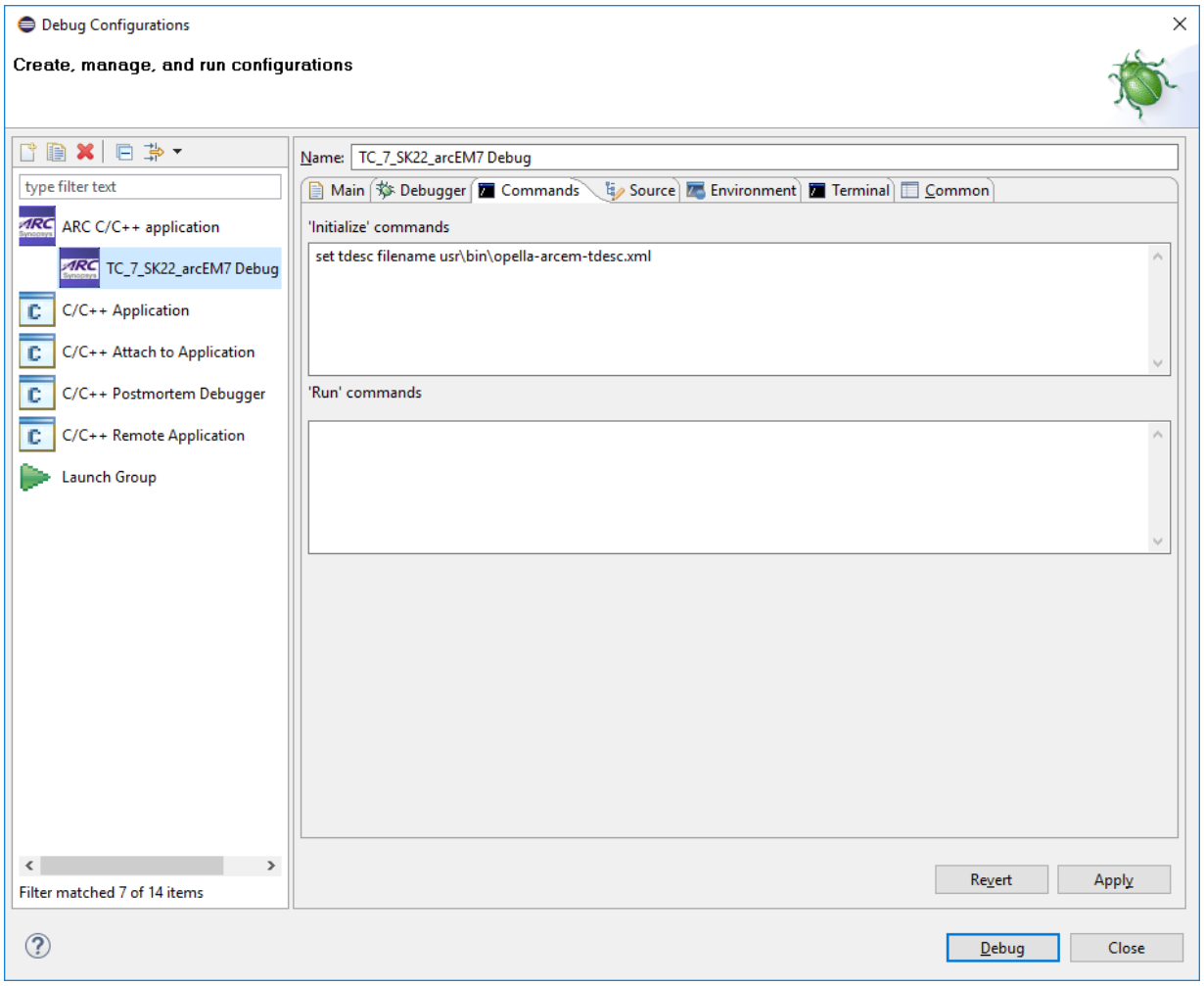
Opella-XD properties



Debugging process with Opella-XD

Alternative way of debugging

Another way to debug an application is connecting to running gdb-server. Run Ashling GDB Server from command line as described in Running Ashling GDB Server. Choose the “Connect to running GDB server” value of ARC GDB Server field on Debugger tab of debug configuration.



On the next tab, Commands, specify path to your tdesc file.

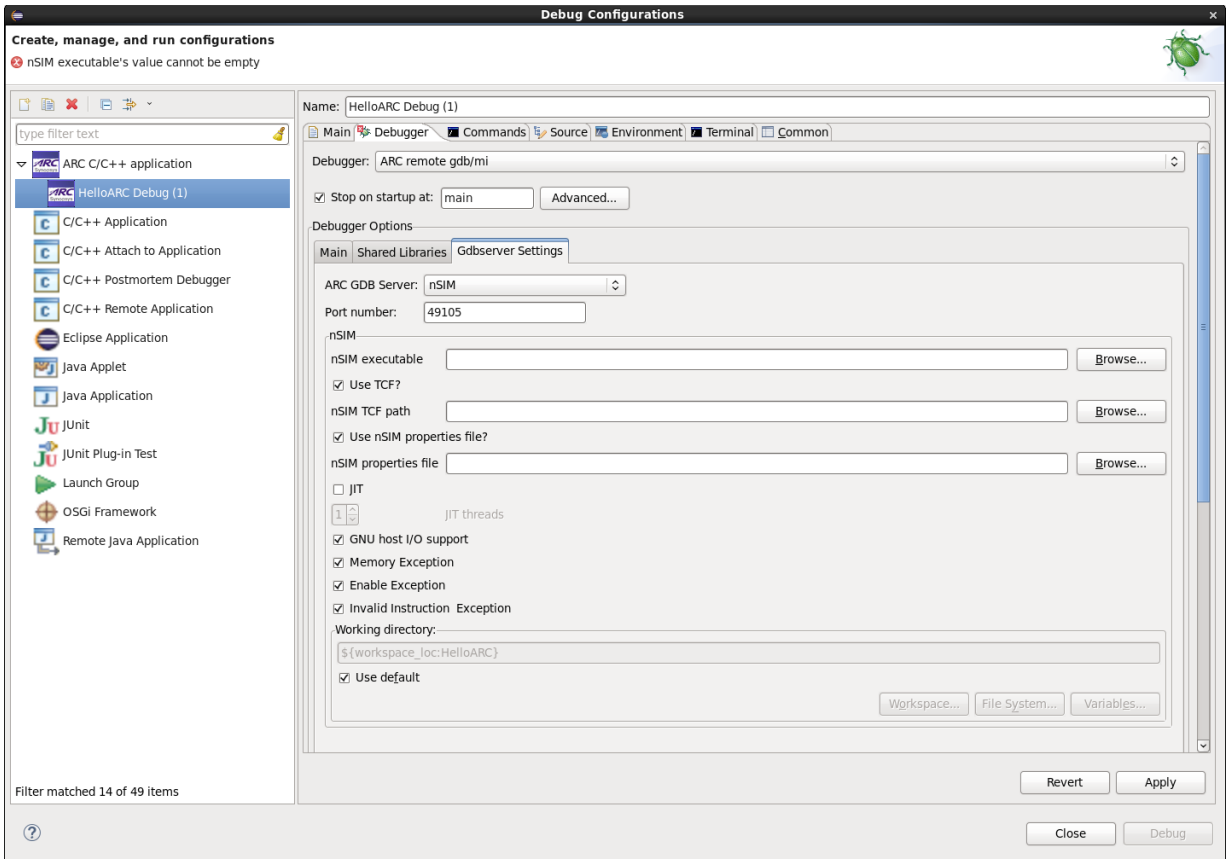
Start debug session after successful initialization of the gdb-server.

Debugging with nSIM

It is expected that you have already built your application and created a debug configuration for it. About how to do it you can read on the following pages:

- Building an Application
- Creating a Debug Configuration

Specifying nSIM properties



Choosing nSIM on debug tab

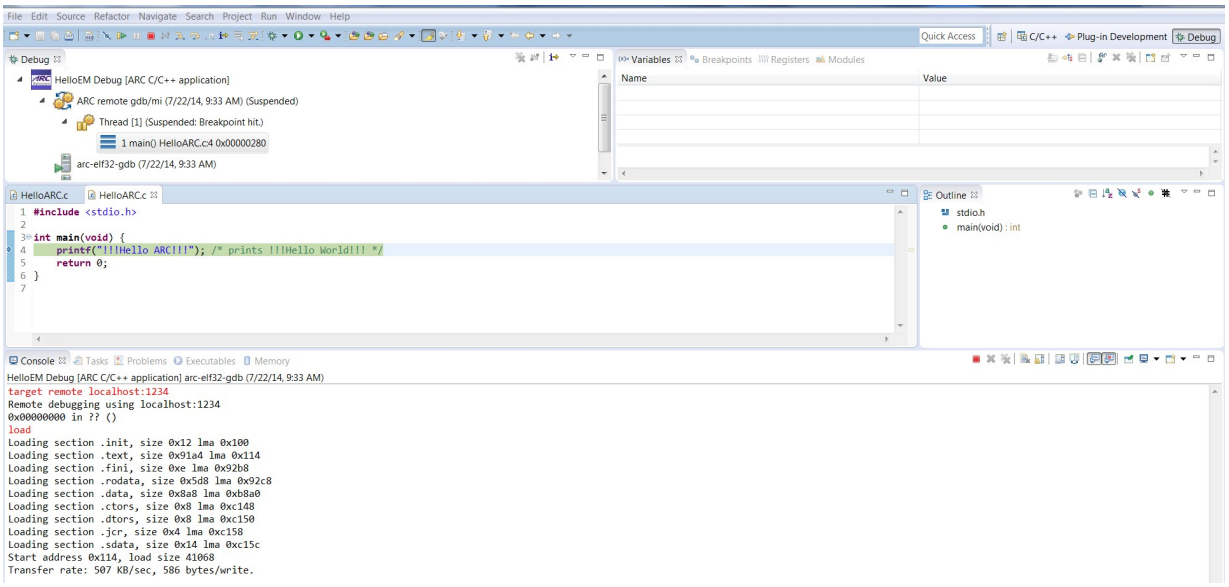
In this tab, user needs to indicate correct properties file/TCF file for current CPU core. In general it is recommended to use TCF files, because they are generated from the Build Configuration Registers and thus most reliably describe target core. nSIM Properties files contain list of key-values for nSIM properties which allow to describe target core and additional simulation features, full list of properties is documented in the nSIM User Guide. It is possible to specify both TCF and properties file, with properties file being able to override parameters set in TCF. For example, if you have a TCF for a little endian core, but would like to simulate it as a big endian, it is possible to create an properties file that will set only a single property for big endian, then in IDE GUI in nSIM GDBserver settings specify paths to both TCF and properties file and that will give a desired results.

Other available options:

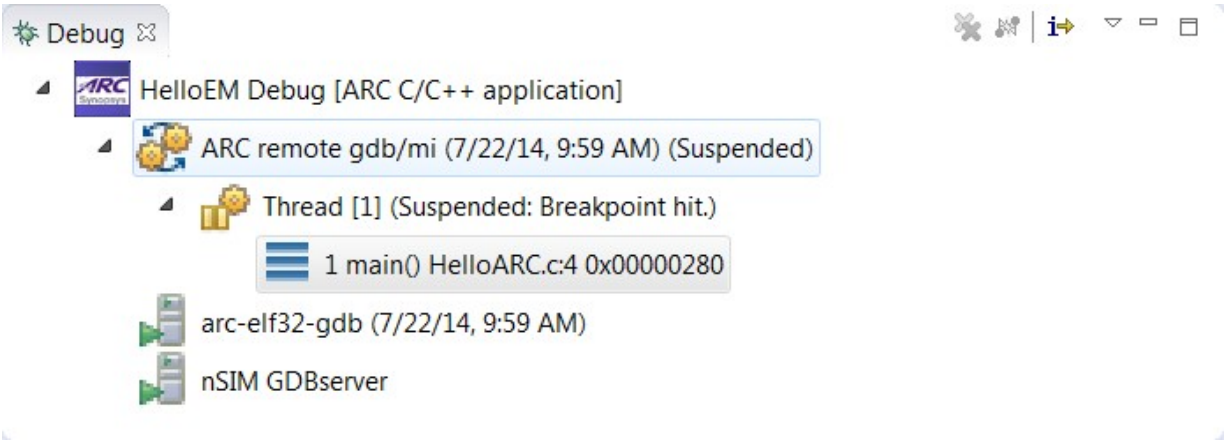
- **JIT** checkbox enables Just-In-Time compilation. You can also specify a number of threads to use in JIT mode.
- **GNU host I/O support**, if checked, enables nSIM GNU host I/O support. It means that input/output requests from application will be handled by nSIM and redirected to the host system. This could be used, for example, to enable functions in the C library, such as `printf()` and `scanf()`. This option works only if the application is built with the ARC GCC compiler and `--specs=nsim.specs` flag is used.
- **Enable Exception**, **Memory Exception** and **Invalid Instruction Exception** options, if checked, tell nSIM to simulate all exceptions, memory exceptions and invalid instruction exceptions, respectively. If one of these options is unchecked and corresponding exception happens, nSIM will exit with an error instead.
- **Working Directory** is a directory from which nSIM GDB server will be started. By default it is project location. This option might be useful if your program works with files. To open a file, you can instead of its absolute path provide a path relative to the specified working directory.

Debugging an application

To debug application using nSIM, press **Debug** button of IDE.



Debugging with nSIM gdbserver



Debug Window

nSIM gdbserver output in console

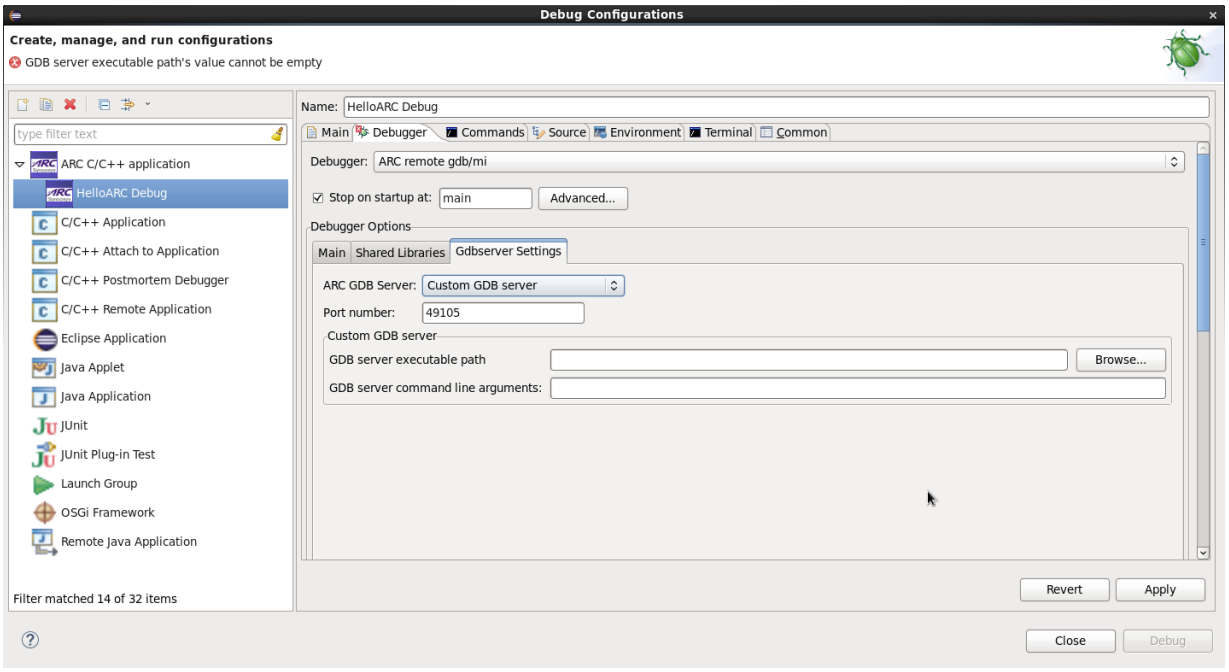
Debugging with Custom GDB Server

It is expected here that you have already built your application and created a debug configuration for it. About how to do it you can read on the following pages:

- Building an Application

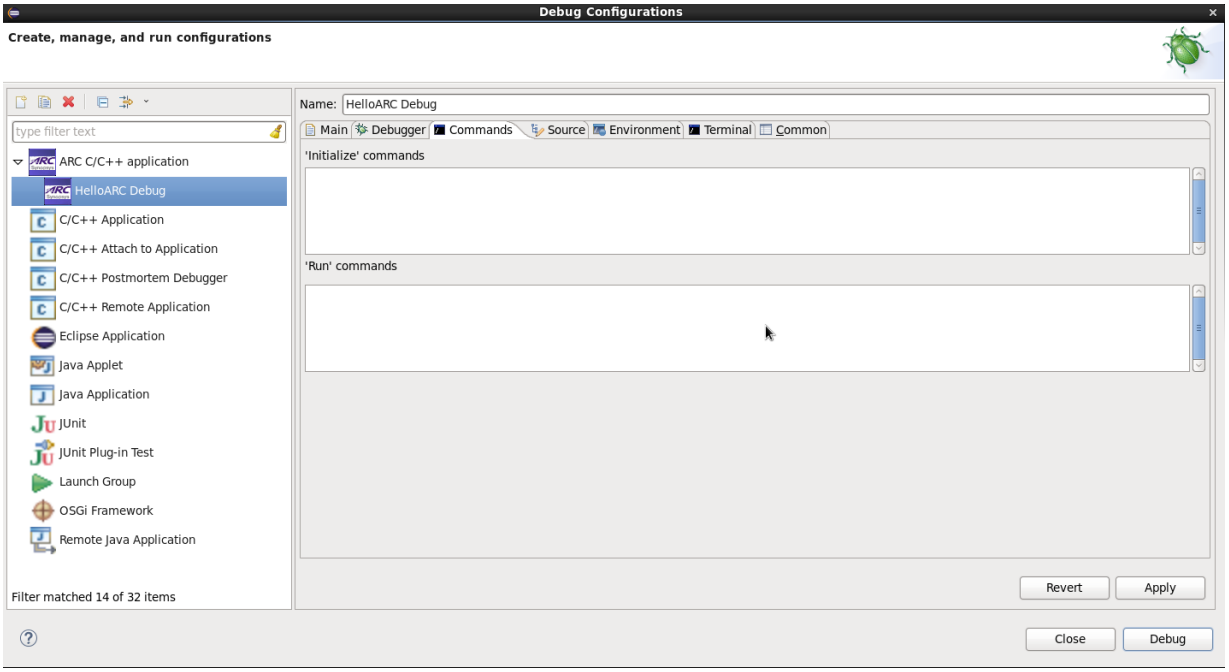
- Creating a Debug Configuration

Specifying custom GDB server properties



Custom GDB Server tab

You can use some other GDB server. In that case you should specify a path to this server executable file, its command-line arguments and also commands to be passed to the GDB client. These are on the **Commands** tab of the dialog.

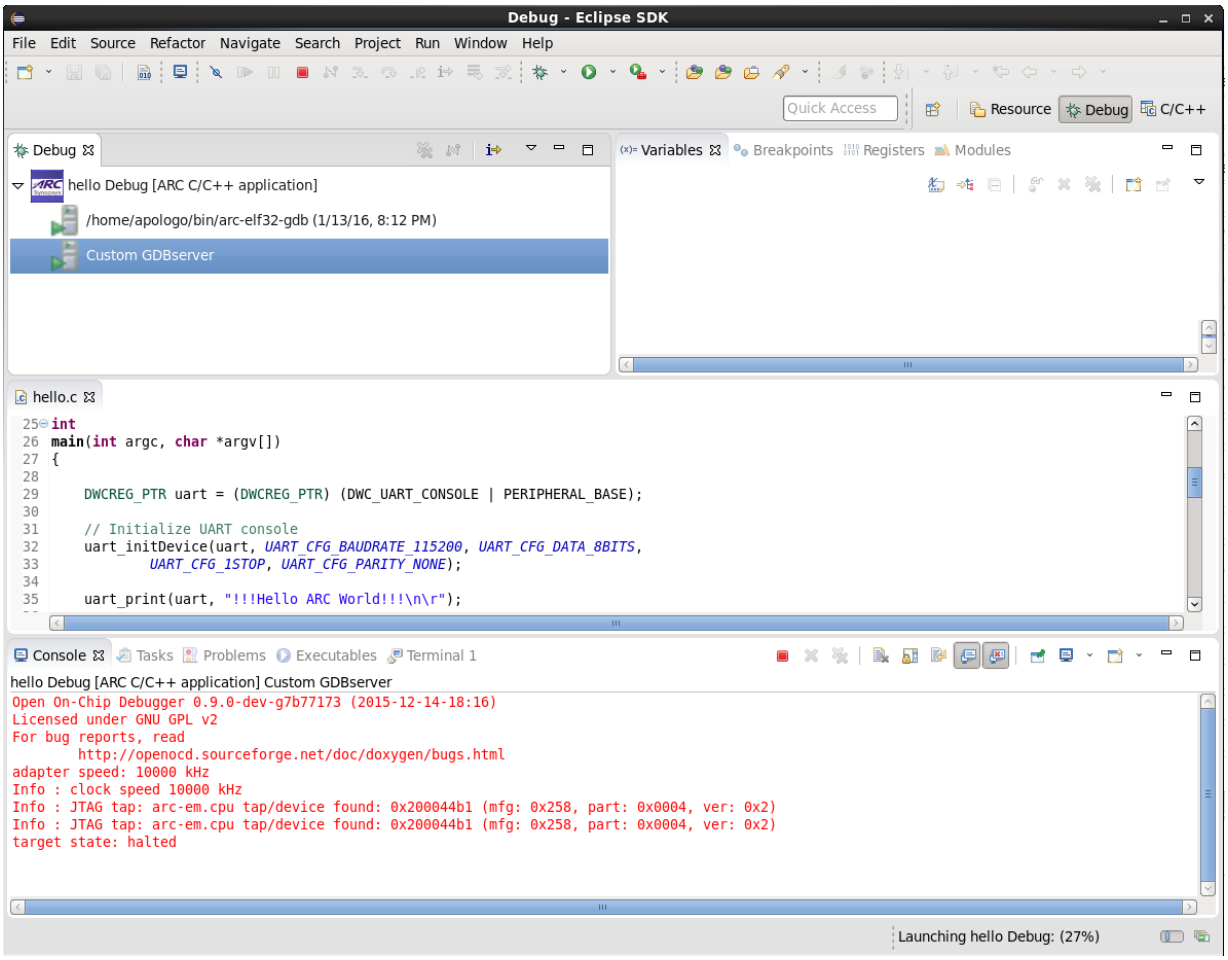


Commands tab

OpenOCD as a custom GDB server

To use OpenOCD as a custom GDB server, user needs to specify command line options for OpenOCD. It is not necessary to specify any commands for GDB on the **Commands** tab, it will connect to OpenOCD automatically.

Custom GDB server properties



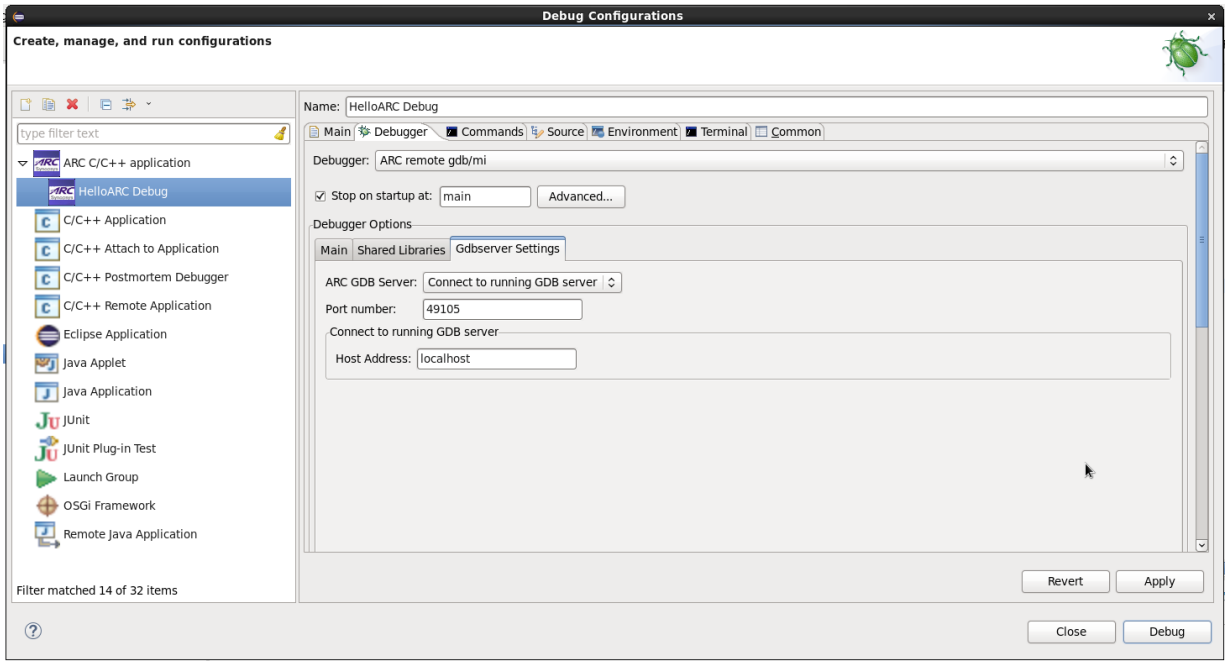
Debugging using custom GDB server

Debugging with Running GDB Server

It is expected here that you have already built your application and created a debug configuration for it. About how to do it you can read on the following pages:

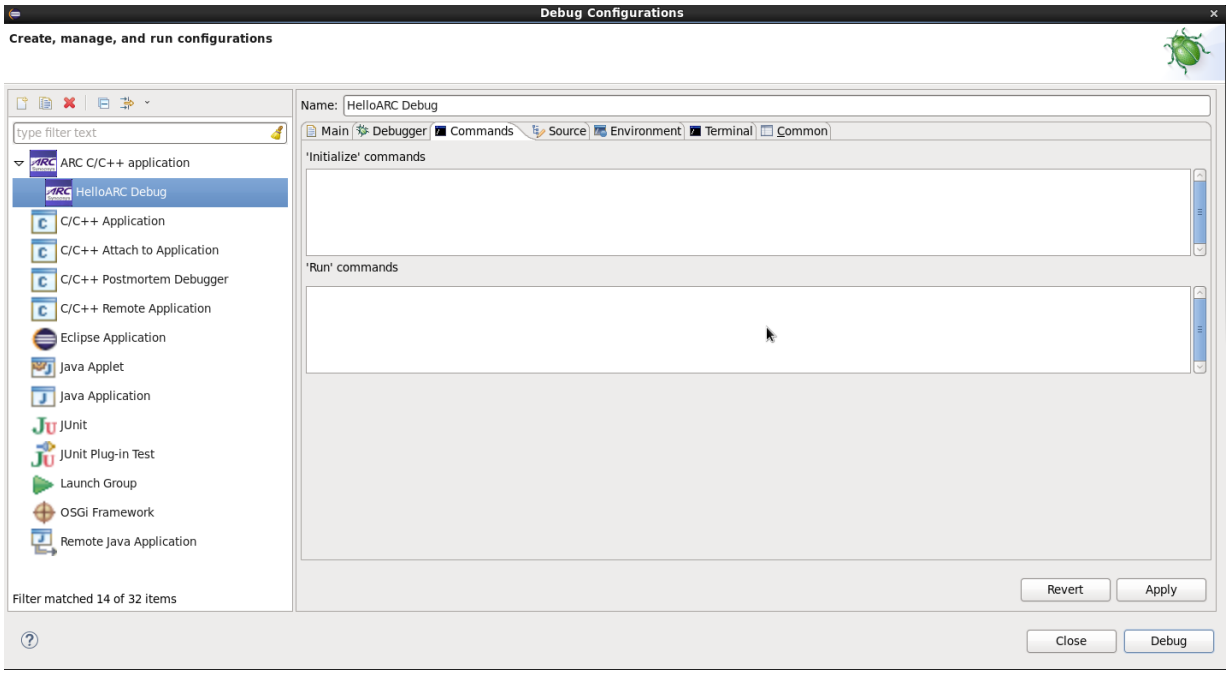
- Building an Application
- Creating a Debug Configuration

Running GDB server properties



Connect to running GDB server tab

If you want to connect to a GDB server that is already running, you should choose a host address and also specify commands to be passed to the GDB client on the **Commands** tab.

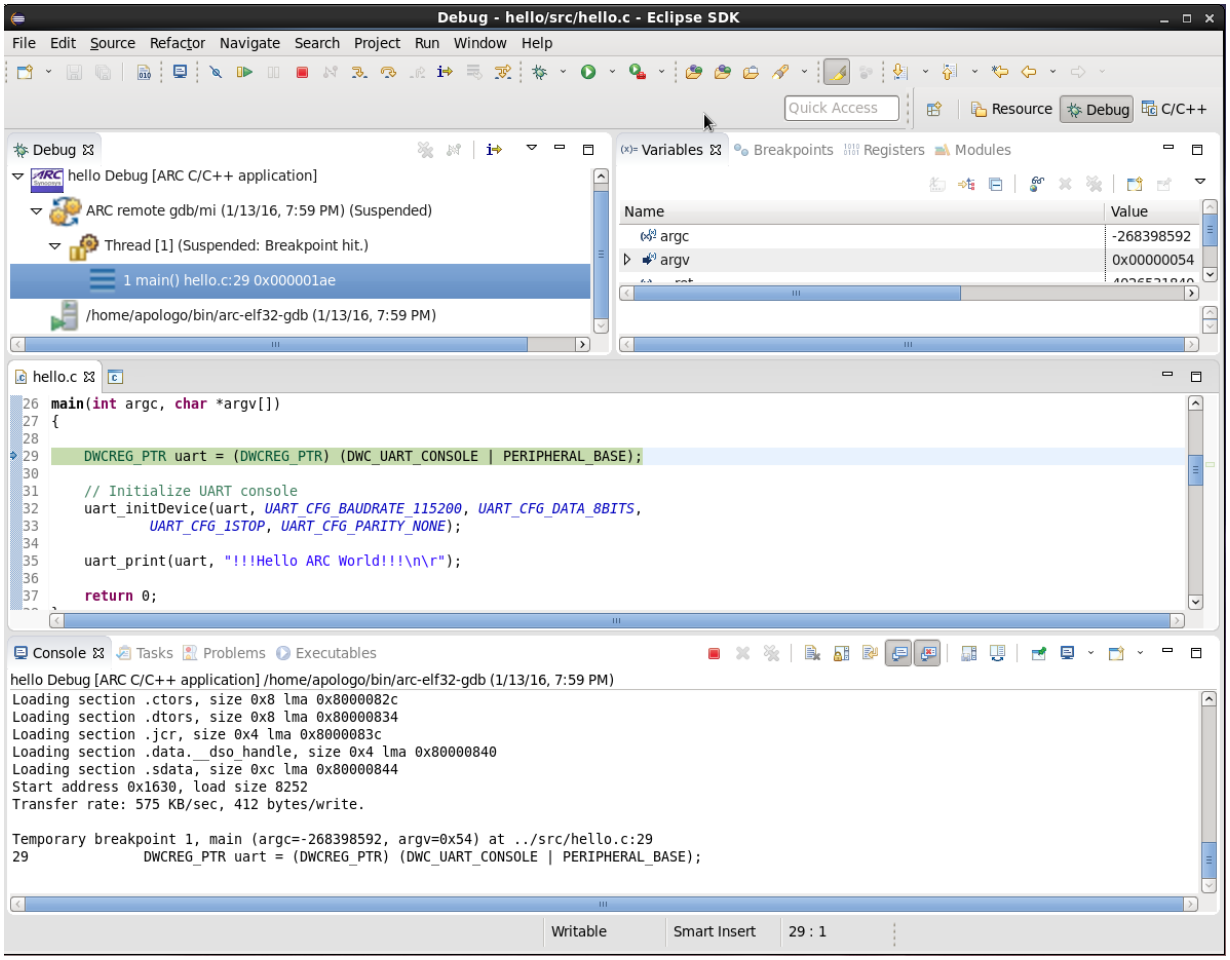


Commands tab

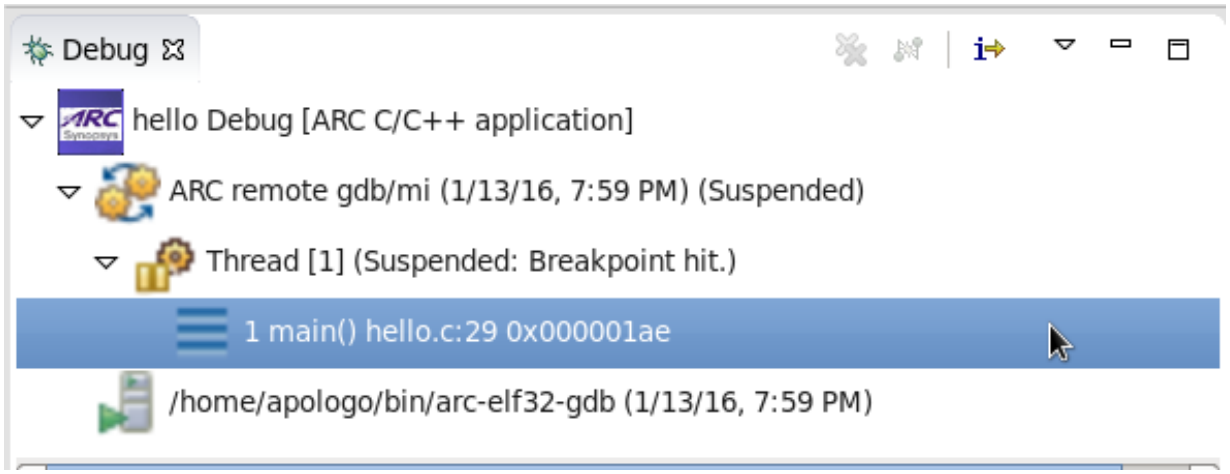
OpenOCD as running GDB server

If you have a running OpenOCD server, you can connect to it just by choosing **Connect to running GDB Server** under **ARC GDB Server** on **Debugger** tab and specifying port number and host address on which your OpenOCD is

running. You do not need to specify any initialize commands for GDB in **Commands tab**, it will connect to OpenOCD using host address and port number from **Debugger** tab.



Debugging using running GDB server



Debug window

```
Open On-Chip Debugger 0.9.0-dev-g7b77173 (2015-12-14-18:16)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.sourceforge.net/doc/doxygen/bugs.html
adapter speed: 10000 kHz
Info : clock speed 10000 kHz
Info : JTAG tap: arc-em.cpu tap/device found: 0x200044b1 (mfg: 0x258, part: 0x0004, ver: 0x2)
Info : JTAG tap: arc-em.cpu tap/device found: 0x200044b1 (mfg: 0x258, part: 0x0004, ver: 0x2)
target state: halted
Info : accepting 'gdb' connection on tcp/49105
Info : dropped 'gdb' connection
Info : accepting 'gdb' connection on tcp/49105
```

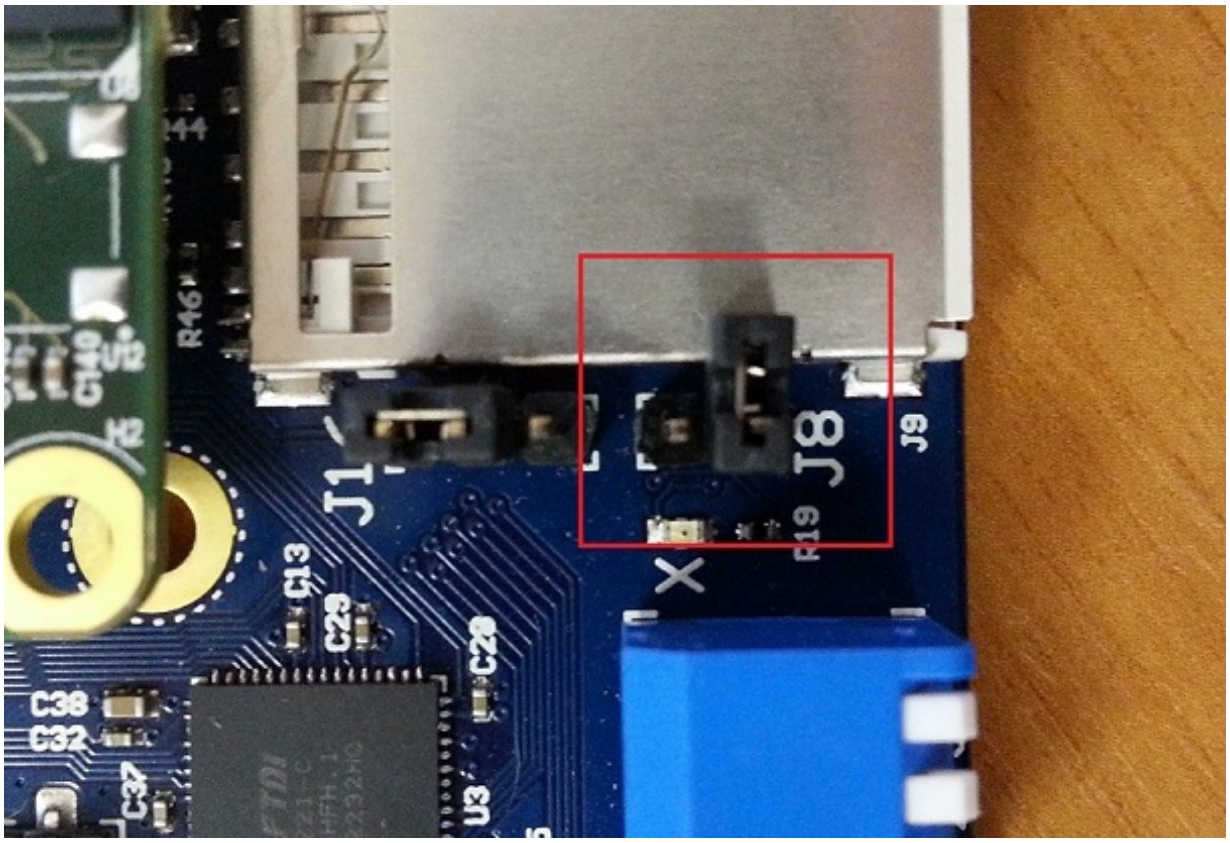
OpenOCD output in console

Debugging a big endian Application on EM Starter Kit

The EM Starter Kit comes with 4 pre-installed little endian configurations. User wishing to work with big endian configuration can use the procedure below to program a big endian .bit file, using the Digilent Adept Software. Big endian .bit file is not a part of the EM Starter Kit Software package.

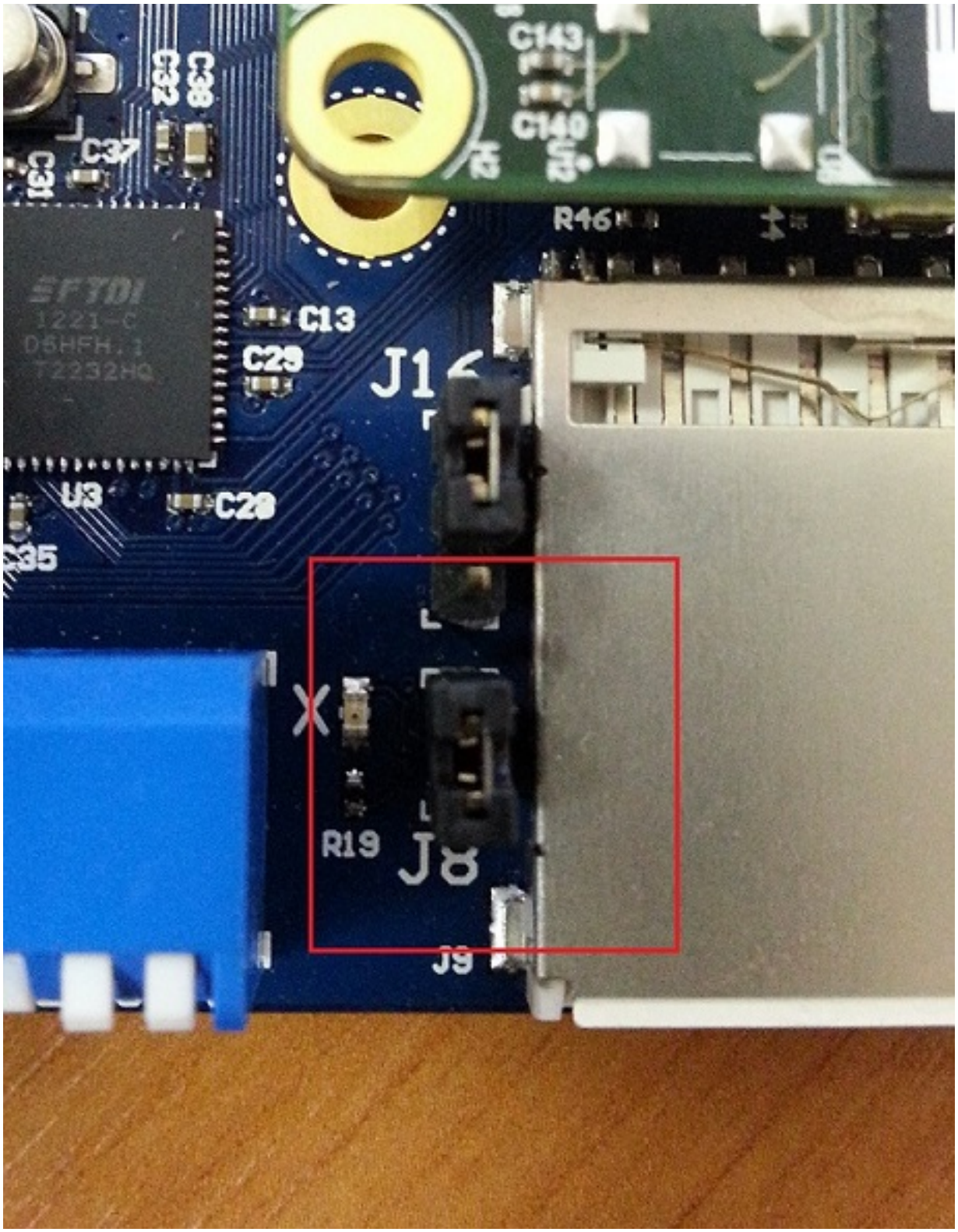
Instruction for Windows

1. Ensure that EM Starter Kit is powered ON and connected to the host PC
2. On the EM Starter Kit, close jumper J8 as shown in images below:



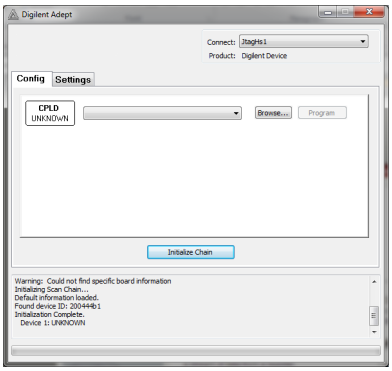
J8 Jumper in factory default position

After closing the jumper:



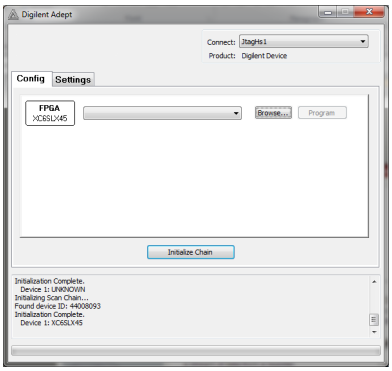
J8 Jumper in closed position

3. Download the Digilent Adept 2 System Software for Windows from http://store.digilentinc.com/digilent-adept-2-download-only/
4. Open the “Adept” utility



Adept Utility before Initializing Chain

5. Press “Initialize chain”. There should be only one device in a chain: XC6SLX150 (XC6SLX45 for ARC EM Starter Kit 1.x)



XC6SLX{150,45} Device shown after Initialization

6. Press “Browse” button and navigate to location of your big endian .bit file
7. Press “Program” button.
8. Return Jumper J8 to its initial position.
9. There are no big endian configuration files for OpenOCD, so to debug your application you should use the same configuration file as for little endian one: `$INSTALL_DIR/share/openocd/scripts/board/snps_em_sk.cfg`, but in the file `$INSTALL_DIR\share\openocd\scripts\target\snps_em_sk_fpga.cfg` replace `-endian little` with `-endian big`.

The EM Starter Kit will now use the selected big-endian FPGA image until the board is powered off or until reconfiguration by pressing the FPGA configuration button located above the “C” in the “ARC” log on the board. Refer to EM Starter Kit documentation for more details.

Instructions for Linux

Follow step 1 through 3 from Windows section to properly configure board and download Adept software. To program FPGA it is required to install both “runtime” and “utilities” packages. After installing utilities and setting jumpers appropriately, use Digilent command-line utilities:

```
$ djtgcfg enum
Found 1 device(s)

Device: TE0604-03
    Product Name:   JTAG-ONB4
    User Name:      TE0604-03
```

```
    Serial Number:  25163300005A

$ djtgcfg init -d TE0604-03
Initializing scan chain...
Found Device ID: 4401d093

Found 1 device(s):
    Device 0: XC6SLX150

$ djtgcfg prog -d TE0604-03 -i 0 -f <bit_file>
Programming device. Do not touch your board. This may take a few minutes...
Programming succeeded.
```

Miscellaneous

Plugin preferences

ARC plugins in IDE support preferences that can alter their behaviour. These preferences are unlikely to be useful for end-user, so they are not exposed via GUI, but they can be modified via preferences files located in Eclipse workspace folder in `.metadata/.plugins/org.eclipse.core.runtime/.settings` folder.

Debugger plugin preferences

Debugger plugin preferences are located in file `com.arc.embeddedcdt.prefs`.

gdbserver_startup_delay

Delay in milliseconds that this plugin wait for after starting gdbserver and before starting the GDB, thus allowing server to start listening on TCP port. Default value is 500.

gdbserver_use_adaptive_delay

Whether to try to use adaptive server startup delay or use only default fixed delay time. Default value is true.

gdbserver_startup_timeout

Amount of time in milliseconds given to gdbserver to start in adaptive startup procedure. Default value is 30000.

gdbserver_startup_delay_step

Amount of time to sleep in milliseconds after adaptive gdbserver startup delay procedure failed to connect to the server. In practice this can be very small, because Socket.connect() itself waits for 1 second. However I've measured this value on my machine, so I'm not sure it is equally valid everywhere, so I leave this is a possible parameter to modify if needed. Default value is 1.

nsim_pass_reconnect_option

Whether to start nSIM with option -reconnect. This is required for adaptive delay to work with nSIM. Default value is true.

Creating Eclipse plugins release zip file

Following is an guide how to create distributable release file for Eclipse for GNU Toolchain for ARC.

1. Create tag for release: `$ git tag arc-2014.12`
2. Clean all past artifacts: `$ git clean -dfx`
3. Ensure that no files are modified: `$ git reset --hard HEAD`
4. Start Eclipse
5. Build plugins. That is important, because even when it is not done, "publishing step" will somehow succeed, but will produce plugins that are only partially functional.

    1. Make sure that "Project / Build Automatically" is checked

2. Go to “Project / Clean”
3. Check “Clean all projects”
4. Press OK button

6. Open `site.xml` file of “ARC GNU Eclipse Update Site”
7. Press “Build All”

8. Zip contents of “updatesite” folder. Note that contents of this folder should be zipped, not the folder itself. Files `.gitignore` and `.project` should be excluded from zip:

```
zip -r arc_gnu_2015.12_ide_plugins.zip artifacts.jar content.jar features/ plugins/ site.xml
```

GCC documentation

This is a collection of information about ARC GCC compiler. It provides various information for how to use efficiently GCC compiler for ARC processors.

ARC specific tech discussion

Custom instructions: what and how

ARC architecture allows users to specify extension instructions. These extension instructions are not macros; the assembler creates encodings for use of these instructions according to the specification by the user.

To create a custom instruction, ones need to make use of the .extInstruction pseudo-op, which also allows the user to choose for a particular instruction syntax, one of:

- Three operand instruction;
- Two operand instruction;
- One operand instruction;
- No operand instruction.

But what is the difference between those ones. To answer this question, we need to check how an extension instruction is encoded:

- Major Opcode = 0x07
  - Sub Opcode1 = 0x00-0x2E, 0x30-0x3f : Used by three operand instructions;
  - Sub Opcode1 = 0x2F:
    - Sub Opcode2 = 0x00-0x3E : Used by two operand instructions;
    - Sub Opcode2 = 0x3F:
      - Sub Opcode3 = 0x00-0x3F: Used by one operand instructions;

The three operand instructions are having op<.cc><.f> a,b,c syntax format, and it is the most general form of an ARC instruction:

- op<.f> a,b,c
- op<.f> a,b,u6
- op<.f> b,b,s12
- op<.cc><.f> b,b,c
- op<.cc><.f> b,b,u6
- op<.f> a,limm,c

- op<.f> a,limm,u6
- op<.f> 0,limm,s12
- op<.cc><.f> 0,limm,c
- op<.cc><.f> 0,limm,u6
- op<.f> a,b,limm
- op<.cc><.f> b,b,limm
- op<.f> a,limm,limm
- op<.cc><.f> 0,limm,limm
- op<.f> 0,b,c
- op<.f> 0,b,u6
- op<.f> 0,limm,c
- op<.f> 0,limm,u6
- op<.f> 0,b,limm
- op<.f> 0,limm,limm

The two operand instructions are having the following syntax format:

- op<.f> b,c
- op<.f> b,u6
- op<.f> b,limm
- op<.f> 0,c
- op<.f> 0,u6
- op<.f> 0,limm

The one operand instructions are having the following syntax format:

- op<.f> c
- op<.f> u6
- op<.f> limm

The no-operand instructions are actually using op<.f> u6 one-operand instruction syntax, with u6 set to zero.

On top of the formal syntax choices, we have also syntax class modifiers:

- OP1_MUST_BE_IMM which applies for SYNTAX_3OP type of extension instructions, specifying that the first operand of a three-operand instruction must be an immediate (i.e., the result is discarded). This is usually used to set the flags using specific instructions and not retain results.
- OP1_IMM_IMPLIED modifies syntax class SYNTAX_2OP, specifying that there is an implied immediate destination operand which does not appear in the syntax. In fact this is actually an 3-operand encoded instruction!

Examples

Example 1

```
.extInstruction insn1, 0x07, 0x2d, SUFFIX_NONE, SYNTAX_3OP|OP1_MUST_BE_IMM
```

will allow us the following syntax:

```
insn1  0,b,c
insn1  0,b,u6
insn1  0,limm,c
insn1  0,b,limm
```

Example 2

```
.extInstruction insn2, 0x07, 0x2d, SUFFIX_NONE, SYNTAX_2OP|OP1_IMM_IMPLIED
```

will allow us the following syntax:

```
insn2  b,c
insn2  b,u6
insn2  limm,c
insn2  b,limm
```

> **Note**
>
> The encoding of insn2 uses the SYNTAX_3OP format (i.e., Major 0x07 and SubOpcode1: 0x00-0x2E, 0x30-0x3F)

Example 3

```
.extInstruction insn1, 7, 0x21, SUFFIX_NONE, SYNTAX_3OP
.extInstruction insn2, 7, 0x21, SUFFIX_NONE, SYNTAX_2OP
.extInstruction insn3, 7, 0x21, SUFFIX_NONE, SYNTAX_1OP
.extInstruction insn4, 7, 0x21, SUFFIX_NONE, SYNTAX_NOP

start:
    insn1   r0,r1,r2
    insn2   r0,r1
    insn3   r1
    insn4
```

will result in the following encodings:

```
Disassembly of section .text:

0x0000 <start>:
   0:   3921 0080               insn1   r0,r1,r2
   4:   382f 0061               insn2   r0,r1
   8:   392f 407f               insn3   r1
   c:   396f 403f               insn4
```

GCC's support for ARC custom extensions

The ARC extension instructions are supported by GNU toolchain by the GNU assembler. These extension instructions are not macros; the assembler creates encodings for use of these instructions according to the specification by the user. To use them at the C-level, we need to make use of inline assembly facility provided by GNU compiler.

Using inline functions

In a header file, define a macro to build a two operand custom instruction:

```
#define intrinsic_2OP(NAME, MOP, SOP)                \
    ".extInstruction " NAME "," #MOP "," \
    #SOP ",SUFFIX_NONE, SYNTAX_2OP\n\t"
```

Now instantiate the extension instruction only once:

```
asm (intrinsic_2OP ("chk_pkt", 0x07, 0x01));
```

Create an inline function:

```
__extension__ static __inline int32_t __attribute__ ((__always_inline__))
__chk_pkt (int32_t __a)
{
  int32_t __dst;
  __asm__ ("chk_pkt %0, %1\n\t"
             : "=r" (__dst)
             : "rCal" (__a));
  return __dst;
}
```

Example:

```
#include <stdint.h>

#define intrinsic_2OP(NAME, MOP, SOP)                                \
  ".extInstruction " NAME "," #MOP "," \
  #SOP ",SUFFIX_NONE, SYNTAX_2OP\n\t"

asm (intrinsic_2OP ("chk_pkt", 0x07, 0x01));

__extension__ static __inline int32_t __attribute__ ((__always_inline__))
__chk_pkt (int32_t __a)
{
  int32_t __dst;
  __asm__ ("chk_pkt %0, %1\n\t"
             : "=r" (__dst)
             : "rCal" (__a));
  return __dst;
}

int foo (void)
{

  return __chk_pkt (10);
}
```

Assembler results:

```
     .file   "t03.c"
     .cpu HS
     .extInstruction chk_pkt,0x07,0x01,SUFFIX_NONE, SYNTAX_2OP

     .section        .text
     .align 4
     .global foo
     .type   foo, @function
foo:
# 13 "t03.c" 1
     chk_pkt r0, 10
```

```
# 0 "" 2
     j_s [blink]
     .size   foo, .-foo
     .ident  "GCC: (ARCompact/ARCv2 ISA elf32 toolchain arc-2016.09-rc1-2-gb04a7b5) 6.2.1 2016082
```

Using a global asm helper file containing the definition of the custom instructions

Define the new assembly instruction in the global assembly helper file (e.g., mycustom.s):

```
.extInstruction chk_pkt, 0x07, 0x01, SUFFIX_NONE, SYNTAX_2OP
```

Define the inline assembly wrapper in a C-source file:

```
#define chk_pkt(src) ({long __dst_;                          \
       __asm__ ("chk_pkt %0, %1\n\t"                         \
              : "=r" (__dst_)                                \
              : "rCal" (src));                               \
          __dst_;})
```

Use the custom instruction:

```
result =chk_pkt(deltachk);
```

Compile,assemble and link it like this (order is important):

Using only defines at the C source level

In a header file, define a macro to build a two operand custom instruction:

```
#define intrinsic_2OP(NAME, MOP, SOP)                        \
    ".extInstruction " NAME "," #MOP ","                     \
       #SOP ",SUFFIX_NONE, SYNTAX_2OP\n\t"
```

Now instantiate the extension instruction only once:

```
__asm__ (intrinsic_2OP ("chk_pkt", 0x07, 0x01));
```

Define a macro for the custom instruction to be used in C sources:

```
#define chk_pkt(src) ({long __dst;                           \
       __asm__ ("chk_pkt %0, %1\n\t"                         \
              : "=r" (__dst)                                 \
              : "rCal" (src));                               \
          __dst;})
```

Use the custom instruction in C-sources:

```
result = chk_pkt(deltachk);
```

Compile,assemble and link it like this:

```
arc-elf32-gcc -O1 foo.c
```

For reference the header file for the above example looks like this:

```
#ifndef _EXT_INSTRUCTIONS_H_
#define _EXT_INSTRUCTIONS_H_

#define intrinsic_2OP(NAME, MOP, SOP)                                          \
    ".extInstruction " NAME "," #MOP "," #SOP ",SUFFIX_NONE, SYNTAX_2OP\n\t"

__asm__ (intrinsic_2OP ("chk_pkt", 0x07, 0x01));

#define chk_pkt(src) ({long __dst;                    \
        __asm__ ("chk_pkt %0, %1\n\t"                 \
          : "=r" (__dst)                              \
          : "rCal" (src));                            \
         __dst;})

#endif /* _EXT_INSTRUCTIONS_H_ */
```

Using the inline assembly can prove difficult if one is using complex instructions. It is recommended to check always if the output/input constrains are matching the instruction definition. In the above example, my assumption is that the custom instruction can access all the “r” registers. If this is not the case, then we should take special care when making the #define(using mov/lr/sr/aex instructions for example). We can also define extension core registers using “.extCoreRegister” assembly directive.

References

- GNU assembler manual from our release: Arc Machine Directives
- GCC -Inline assembly -Howto

Impact of 64-bit integral operation on GCC toolchain

Intro

Some of the new 64 bit integral operations made available for ARCv2HS can be used to map the C-type long long. These are:

Operations	Hardware option	Possible compiler usage
LDD/STD	LL64_OPTION	Load/store 64 bit data type
Chained MPYM/MPYMU	MPY_OPTION_{5, 6}	Implementation of 32x32->64 bit ops
MAC/MACU	MPY_OPTION_7	Multiply and accumulate operations
MACD/MACDU	MPY_OPTION_8+	Multiply and accumulate operations
MPYD/MPYDU	MPY_OPTION_8+	Implementation of 32x32->64 bit ops
VADD2	MPY_OPTION_9	Register to register move of a 64 bit data type

64-bit move operations

First step in efficiently supporting the long long data type is implementing an efficient way to move the 64 bit data type in and out register file as well as within register file. The LL64_OPTION provides us with the means for fast transfer of 64 bit data into a processor register pair. The LDD/STD can be used as well to implement a fast way to save/restore the registers in prologue/epilogue of a function.

The MPY_OPTION_9 also gives us means to move a register to another register or a 32-bit immediate into a 64 bit register. The 32-bit immediate is signed extended to match the 64 bit container. Hence, for a register to register move, we can use the following instruction:

```
VADD2        r0r1,r2r3,0
```

The above instruction takes 32 bits in the program memory as it uses the VADD2 A,B,u6 encoding. Although VADD2 supports predication, we cannot use it for register to register move due to ISA limitations (e.g., the source of the operands needs to be the input argument vadd2 .cc b,b,u6) If we want to move and sign extend a 32-bit immediate into a 64-bit register pair, we can use the following instruction:

```
VADD2        r0r1, 0xAFEF, 0
```

The above instruction takes 64 bits in the program memory as we use VADD2 A,limm,u6 encoding.

Multiplication Instructions

The implementation of multiplication instructions depends on the multiplier option used. A special care should be taken for chained operation when MPY_OPTION is either 5 or 6. In these configurations, the multiplier is blocking sequential, hence, the chained option improves the multiplication result. This, however, may be relevant for EM series as the HS will employ a fully pipelined multiplier.

In general, for 32x32bit -> 64 bit type of multiplier, we use the {mpy,mpym } instructions pair. However, when using MPY_OPTION larger than 7, we can make use of the MPYD/MPYDU instructions. These instructions are faster and are having a smaller impact on memory size than previous used solution. Please remark that the MPYD/MPYDU clobbers also the 64-bit accumulator register (ACCH,ACCL).

Multiply and Accumulate instructions

The ISAv2, provides a number of MAC operations. These are MAC/MACU for MPY_OPTION equals to 7, and additionally MACD/MACDU when using MPY_OPTION eight or more. The latter ones are interesting as they place the 64 bit result in a register pair. All the MAC operations are using the 64-bit accumulator register (ACCH,ACCL) to accumulate with, as well to place the result mac into.

Using a MAC operation needs to set up the accumulator register, as well as collecting the result from the accumulator and place it into a general purpose register. Hence,

Used instructions	Single MAC (instructions)	Multiple MACs, unroll case	Throughput
MAC / MACU	4 (2 loads into ACCH,ACCL; 1 MAC; 1 move from ACCH to register)	4 + 1 for each unrolled MAC (2 to initialize ACCH,ACCL; 2 to move the accumulator)	3+ (output/anti-dependency on ACC), 1 (otherwise)
MACD / MACDU	3 (2 loads to ACCH,ACCL; 1 MAC)	2 + 1 for each unrolled MAC	3+ (output/anti-dependency on ACC), 1 (otherwise)
ADD / MPYD	3 (2 additions; 1 MPYD)	3 ops for each MAC	3
ADD / MPY	4 (2 additions; 2 multiplications)	4 ops for each MAC	4

Caveats

Having the implicit 64-bit accumulator as destination for MPYD/MPYDU operations complicate the generated code when we have an anti-dependency with a MAC operation on the accumulator register.

The accumulator register is used as input as well as output for the MAC operation, hence, using them in a pipelined fashion may be difficult (if, for example, between mac operations exist an output/anti-dependency). In this case, it is faster to use an implementation with ADD/MPYD operations.

Case study

Let us consider the following C-program:

```
long long foo (long long a, int b, int c)
{
  a += (long long )c * (long long )b;
  return a;
}
```

Implementation	Resulted Code (estimated)

ADD/MPY	`mpym r5,r3,r2`
	`mpy r4,r3,r2`
	`add.f r0,r0,r4`
	`adc r1,r1,r5`

ADD/MPYD	`mpyd r2,r3,r2`
	`add.f r0,r2,r0`
	`adc r1,r3,r1`

MAC	`mov ACCL,r0`
	`mov ACCH,r1`
	`mac r0,r2,r3`
	`mov r1,ACCH`

MACD (option 8)	`mov ACCL,r0`
	`mov ACCH,r1`
	`macd r0,r2,r3`

MACD (option 9)	`vadd2 ACC,r0,0` `macd r0,r2,r3`

Implementation matrix used by GCC

Due to the accumulator caveats, I propose the following implementation matrix for MAC ops:

MPY_OPTION	2	3	4	5	6	7	8	9
ADD/MPY	Y	Y	Y	Y	Y	Y	N	N
ADD/MPYD	N	N	N	N	N	N	Y	N
MAC	N	N	N	N	N	N	N	N
MACD	N	N	N	N	N	N	N	Y

Using JLI Instructions with GNU

The ARCv2 ISA provides the JLI instruction, which is two-byte instructions that can be used to reduce code size for an application. To make use of it, we provide two new function attributes `jli_always` and `jli_fixed` which will force the compiler to call the indicated function using a jli_s instruction. The compiler also generates the entries in the JLI table for the case when we use `jli_always` attribute. In the case of `jli_fixed` the compiler assumes a fixed position of the function into JLI table. Thus, the user needs to provide an assembly file with the JLI table for the final link. This is useful when we want to have a table in ROM and a second table in the RAM memory.

The jli instruction usage can be also forced without the need to annotate the source code via `-mjli-always` command.

Optimizing Code using JLI Calls on Functions

The usual way of using jli calls is to use the attribute _jli_always_ with a function. For example:

```
int func (int i) __attribute__((jli_always));

int func (int i)
{
  return i*i;
}

int main ()
{
  printf ("func returned = %d \n", func (100));
  return 0;
}
```

which leads to:

```
main:
        push_s blink
        st.a fp,[sp,-4] ;28
        mov_s fp,sp     ;4
        mov_s r0,100    ;3
        jli_s @__jli.func
        mov_s r2,r0     ;4
        mov_s r1,r2     ;4
        mov_s r0,@.LC0  ;14
        bl @printf;1
        mov_s r2,0      ;3
        mov_s r0,r2     ;4
        ld.ab fp,[sp,4] ;25
        pop_s blink
        j_s [blink]
```

As we can see the call to func is done via the `jli_s` instruction, while the other calls are done using regular `bl` instruction. If we want all calls to non-local functions to be done using jli instructions we can use `-mjli-always` compiler option. However, we need to be careful in using this option as the JLI table can hold only 1024 entries. The compiler cannot efficiently check the number of entries as it only has a limited view over the whole application. In this case the GNU tool takes care of generating the JLI table, patching the `jli_s` instruction with the correct entry number corresponding to the called function, and the initialization of the jli_base auxiliary register.

A special way to use the jli instruction is for ROM patching. Because with the jli instruction function calls are made indirectly through the JLI table, the JLI table entries can be changed to invoke alternative functions without affecting the executable code. Thus, in this case the location of each function called via jli instruction must be fixed and known at compile time. To achieve this, we have introduced a new `jli_fixed` function attribute which accept a numerical parameter to specify the function call entry in the JLI table. This attribute is GNU specific.

Let us consider the following example:

```
int func (int i) __attribute__((jli_fixed(2)));

int func (int i)
{
  return i*i;
}

int main ()
{
  printf ("func returned = %d \n", func (100));
  return 0;
}
```

which leads to:

```
main:
        push_s blink
        st.a fp,[sp,-4] ;28
        mov_s fp,sp     ;4
        mov_s r0,100    ;3
        jli_s 2 ; @func
        mov_s r2,r0     ;4
        mov_s r1,r2     ;4
        mov_s r0,@.LC0  ;14
        bl @printf;1
        mov_s r2,0      ;3
        mov_s r0,r2     ;4
        ld.ab fp,[sp,4] ;25
        pop_s blink
        j_s [blink]
```

As we can see now, the operand of jli instruction is already resolved and points to entry 2 in the JLI table. In this case, the compiler doesn't generate the JLI table, as it needs to be provided by the user. A JLI table can be something like this:

```
        .section .jlitab
        .align  4
JLI_table:
__jli.entry0:   b       entry0  ; 0
__jli.entry1:   b       entry1  ; 1
__jli.func:     b       func    ; 2
```

The initialization of the jli_base is again done by the crt0. However, in the case of RAM/ROM patching, one may want to overwrite the initial value with a new value based on the location of a patched JLI table. N.B. the RAM/ROM patching approach may require special startup and/or linker scripts which are not provided.

Discussion about MWDT/GNU Compatibility

In general the GNU jli implementation is compatible with MWDT implementation, except for the code that invokes the MetaWare runtime initialization code that sets the JLI_BASE register to address the JLI table. GNU additionally introduces the `jli_fixed` attribute to closely mimic the MWDT `jli_call_fixed` pragma.

SecureShield Programming

SecureShield API

A secure-callable API is a set of functions executing in secure mode that can be called from code executing in normal mode. The processor contains a special function call instruction, SJLI, that the compiler uses to implement a secure-mode API. This instruction transfers execution from normal mode to secure mode. Any other call or jump into secure mode from normal mode results in a processor exception.

The GNU tools support a secure-callable API with a two-executable approach:

- One linked executable contains the secure-mode code;
- The second linked executable contains the normal-mode code.

The normal-mode code is compiled normally—no special code is generated for calls to the secure-mode API. However, such calls are resolved by the linker in the normal executable with special function entry points that transfer control to the normal executable using the SJLI instruction. In the secure-mode executable, functions that are designate as belonging to the secure API need an index into the SJLI table. Also the runtime initialization for the secure-mode needs to be carried on by the user.

Identifying the Secure-Callable APIs

To indicate to the compiler that a secure-mode function is callable from normal mode, you can use `__attribute__((secure_call (IndexNumber)))` with secure-callable function. Where `IndexNumber` is the entry of that particular function into the SJLI table.

Programming Cautions

Using function pointer of a secure call function is not supported. However, one can make a stub which can be called indirectly, the stub itself calls the secure call normally.

Example

Let us consider the following example:

```
#include <stdio.h>

extern int foo (int) __attribute__((secure_call(2)));

int bar (void)
{
  printf ("%d\n", foo (100));
  return 0;
}

int bar2 (void)
{
  return foo(100);
}
```

Where function `foo()` is an external secure function located at index 2 in the SJLI table.

The result is:

```
.cpu EM
.section     .rodata.str1.4,"aMS",@progbits,1
.align 4
      .LC0:
         .string     "%d\n"
         .section    .text
         .align 4
         .global     bar
         .type       bar, @function
      bar:
         push_s blink
         mov_s r0,100        ;3
         sjli  2     ; @foo
         mov_s r1,r0 ;4
         mov_s r0,@.LC0      ;14
         bl @printf;1
         pop_s blink
         j_s.d [blink]
         mov_s r0,0  ;3
         .size       bar, .-bar
         .align 4
         .global     bar2
         .type       bar2, @function
      bar2:
         push_s blink
         mov_s r0,100        ;3
         sjli  2     ; @foo
         pop_s blink
         j_s [blink]
          .size      bar2, .-bar2
```

Where, we can easily spot the call to `foo()` function via `SJLI` instruction.

Sanitizers support for ARC

Sanitizers functionality has been enabled for ARC.

Please notice that support for the sanitizers is limited for Linux, when running with upcoming glibc (GNU C Library) port.

ARC supports the following sanitizers:

- Address,
- Memory,
- Undefined behavior and
- Leak sanitizers.

AddressSanitizer is a fast memory error detector. It consists of a compiler instrumentation module and a run-time library. The tool can detect the following types of bugs:

- Out-of-bounds accesses to heap, stack and globals
- Use-after-free
- Use-after-return (runtime flag ASAN_OPTIONS=detect_stack_use_after_return=1)
- Use-after-scope (clang flag -fsanitize-address-use-after-scope)
- Double-free, invalid free
- Memory leaks (experimental)

Typical slowdown introduced by AddressSanitizer is 2x.

MemorySanitizer is a detector of uninitialized reads. It consists of a compiler instrumentation module and a run-time library. Typical slowdown introduced by MemorySanitizer is 3x.

UndefinedBehaviorSanitizer (UBSan) is a fast undefined behavior detector. UBSan modifies the program at compile-time to catch various kinds of undefined behavior during program execution, for example:

- Using misaligned or null pointer
- Signed integer overflow
- Conversion to, from, or between floating-point types which would overflow the destination
- See the full list of available checks below.

UBSan has an optional run-time library which provides better error reporting. The checks have small runtime cost and no impact on address space layout or ABI.

LeakSanitizer is a run-time memory leak detector. It can be combined with AddressSanitizer to get both memory error and leak detection, or used in a stand-alone mode. LSan adds almost no performance overhead until the very end of the process, at which point there is an extra leak detection phase.

We included in this document a few example as reference to the use of the sanitizers. For a clearer view of how to use the sanitizers, or further examples, please refer to Clang documentation

Address Sanitizer Examples

Heap-use-after-free

```
// To compile: clang++ -O -g -fsanitize=address heap-use-after-free.cc
int main(int argc, char **argv) {
```

```
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

```
$ ./a.out
==5587==ERROR: AddressSanitizer: heap-use-after-free on address 0x61400000fe44 at pc 0x47b55f bp
 sp 0x7ffc36b281f8
READ of size 4 at 0x61400000fe44 thread T0
    #0 0x47b55e in main /home/test/example_UseAfterFree.cc:7
    #1 0x7f15cfe71b14 in __libc_start_main (/lib64/libc.so.6+0x21b14)
    #2 0x47b44c in _start (/root/a.out+0x47b44c)

0x61400000fe44 is located 4 bytes inside of 400-byte region [0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x465da9 in operator delete[](void*) (/root/a.out+0x465da9)
    #1 0x47b529 in main /home/test/example_UseAfterFree.cc:6

previously allocated by thread T0 here:
    #0 0x465aa9 in operator new[](unsigned long) (/root/a.out+0x465aa9)
    #1 0x47b51e in main /home/test/example_UseAfterFree.cc:5

SUMMARY: AddressSanitizer: heap-use-after-free /home/test/example_UseAfterFree.cc:7 main
Shadow bytes around the buggy address:
  0x0c287fff9f70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fa0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c287fff9fc0: fa fa fa fa fa fa fa[fd]fd fd fd fd fd fd fd
  0x0c287fff9fd0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9fe0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9ff0: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c287fffa000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:     fa
  Heap right redzone:    fb
  Freed heap region:     fd
  Stack left redzone:    f1
  Stack mid redzone:     f2
  Stack right redzone:   f3
  Stack partial redzone: f4
  Stack after return:    f5
  Stack use after scope: f8
  Global redzone:        f9
  Global init order:     f6
  Poisoned by user:      f7
  ASan internal:         fe
==5587==ABORTING
```

Heap-buffer-overflow

```
// RUN: clang++ -O -g -fsanitize=address %t && ./a.out
int main(int argc, char **argv) {
      int *array = new int[100];
      array[0] = 0;
      int res = array[argc + 100];  // BOOM
      delete [] array;
      return res;
}
```

```
==25372==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61400000ffd4 at pc 0x0000004c
READ of size 4 at 0x61400000ffd4 thread T0
    #0 0x46bfee in main /tmp/main.cpp:4:13

0x61400000ffd4 is located 4 bytes to the right of 400-byte region
[0x61400000fe40,0x61400000ffd0)
allocated by thread T0 here:
    #0 0x4536e1 in operator delete[](void*)
    #1 0x46bfb9 in main /tmp/main.cpp:2:16
```

Stack-buffer-overflow

```
// RUN: clang -O -g -fsanitize=address %t && ./a.out
int main(int argc, char **argv) {
      int stack_array[100];
      stack_array[1] = 0;
      return stack_array[argc + 100];  // BOOM
}
```

```
==7405==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff64740634 at pc 0x46c103 b
READ of size 4 at 0x7fff64740634 thread T0
    #0 0x46c102 in main /tmp/example_StackOutOfBounds.cc:5

Address 0x7fff64740634 is located in stack of thread T0 at offset 436 in frame
    #0 0x46bfaf in main /tmp/example_StackOutOfBounds.cc:2

  This frame has 1 object(s):
       [32, 432) 'stack_array' <== Memory access at offset 436 overflows this variable
```

Global-buffer-overflow

```
// RUN: clang -O -g -fsanitize=address %t && ./a.out
int global_array[100] = {-1};
int main(int argc, char **argv) {
      return global_array[argc + 100];  // BOOM
}
```

```
==7455==ERROR: AddressSanitizer: global-buffer-overflow on address 0x000000689b54 at pc 0x46bfd8
READ of size 4 at 0x000000689b54 thread T0
    #0 0x46bfd7 in main /tmp/example_GlobalOutOfBounds.cc:4

0x000000689b54 is located 4 bytes to the right of
  global variable 'global_array' from 'example_GlobalOutOfBounds.cc' (0x6899c0) of size 400
```

References

- Wikipedia
- Clang documentation

Working with small data section

> **Note**
>
> These notes are applicable to ARC GCC starting only with 2017.09 release.

Small data available access range

Data Type	Range	#Elements	Size
char	[-256,255]	512	512 bytes
short	[-256,510]	383	766 bytes
int	[-256,1020]	319	1276 bytes

The lower limit depends on the possibility to access byte-aligned datum, hence, it is hard connected to the range of s9 short immediate (i.e., -256). Any other access can be done using address-scaling feature of the load/store instructions.

The number of elements which we can fit in sdata section highly depends on the data alignment properties. For example if we use only 4 byte datum, 1 byte aligned, we can fit up to 128 elements in the section.

Sdata and address scaling mode discussion

To increase the access range for small data area, the compiler will use scaled addresses whenever it is possible. Thus, we can extend theoretically this range to [-1024,1020] for 32-bit data (e.g., _int_) if type aligned (i.e., 4-bytes). However, as we cannot be 100% sure we address only 32-bit/4-byte aligned data, we need to consider the worst case which is byte-aligned data. Thus, the effective range is [-256,255], with possibilities to access 16-bit aligned data (e.g., _short_) up to 510, and 32-bit aligned data (e.g., `int` or `long long`) up to 1020. While the lower limit remains set to -256. This is because we set the linker script defined variable `__SDATA_BEGIN__` with an offset of 0x100. However, this rule can be overwritten by using a custom linker script.

Controlling what goes into SDATA segment

Automatic for global data

Global data smaller than a given number in bytes can be placed into the sdata section. The number of bytes can be controlled via **-G<number>** option.

For ARC, by default this number is set to 8 whenever we have double load/store operations available (i.e., ARC HS architecture), otherwise to 4.

For example, a 8 bytes setting will allow us to place into sdata the following variables:

```
char gA[8];
short gB[4];
int gC[2];
long long gD;
```

Notable exceptions:

- Volatile global data will not be placed into sdata section when **-mno-volatile-cache** option is used;
- Strings and functions never end in small data area;
- Weak variables as well not;
- No constant will end in small data area as those one, we would like to place into ROM.

Using named sections

Another way to control which data goes into small data area is to use named sections like this:

```
int a __attribute__((section (".sdata"))) = 1;
int b __attribute__((section (".sbss")));
```

Variables _a_ and _b_ will go into sdata/sbss sections without checking the data type size against the **-G<number>** value. Thus, we can always control which data is accesses via _gp_ register by setting **-G0** and using named sections to the desired ones. Using named section we can also place into sdata static data.

From MWDT to GCC

MWDT vs. GCC option matrix

Option	ARChitect	CCAC	GCC	MDB	Supported core	Notes
Processor Family	ARC600	-a6	-mcpu=arc600	-a6	ARC600	
	ARC700	-a7	-mcpu=arc700	-a7	ARC700	
	ARCv2EM	-av2em	-mcpu=arcem	-av2em	ARC EM	
	ARCv2HS	-av2hs	-mcpu=archs	-av2hs	ARC HS	
ARC600 Core Versions	Version 1	-core1	N.A.	-core1	ARC600	Initial version
	Version 2	-core2	N.A.	-core2	ARC600	Zeroed BCR Region 0xc0
	Version 3	-core3	N.A.	-core3	ARC600	LD/ST Queue changes
	Version 4	-core4	N.A.	-core4	ARC600	SYNC instruction
	Version 5	-core5	N.A.	-core5	ARC600	ARC600_BUILD BCR
	Version 6	-core6	N.A.	-core6	ARC600	Misaligned LD/ST traps.
ARC700 Core Versions	Version 1	-core1	N.A.	-core1	ARC700	Initial version
	Version 2	-core2	N.A.	-core2	ARC700	Zeroed BCR Region 0xc0
	Version 3	-core3	N.A.	-core3	ARC700	MXP Debug Architecture
	Version 4	-core4	N.A.	-core4	ARC700	SWAPE, LLOCK,SCOND instr
ARC EM Core Versions	Version 0	-core0	Not supported	-core0	ARC EM	EM 1.0 (Initial version)
	Version 1	-core1	N.A.	-core1	ARC EM	EM 1.1 (Default version)

Option	ARChitect	CCAC	GCC	MDB	Supported core	Notes
ARC HS Core Versions	Version 0	-core0	Not supported	-core0	ARC HS	HS 1.0 (Initial version)
	Version 1	-core1	N.A.			
Shift ISA Extension (shift_option)	Option 1	-Xsa	N.A.	-Xsa (-Xshift_assist)		
	Option 2	-Xbs	N.A.	-Xbs (-Xbarrel_shifter)		
	Option 3	-Xsa	-mbarrel-shifter	-Xsa -Xbs	All	Default ON
Code Density Extension	code_density_option	-Xcd	-mcode-density	-Xcd (-Xcode_density)	ARC EM, ARC HS	Default for HS, EM
Bitscan ISA Extension	bitscan_option	-Xnorm	-mnorm	-Xnorm	All	Default for HS
SWAP ISA Extension	swap_option	-Xswap	-mswap	-Xswap	All	Default for HS
DIV_REM ISA option	div_rem_option	-Xdiv_rem	-mdiv-rem	-Xdiv_rem	ARC EM, ARC HS	radix2 is default for EM and HS
Multiply ISA Option (mpy_option)	wlh1	-Xmpy16	-mmpy-option=16	-Xmpy16	ARC600, ARC EM	
	wlh2	-Xmpy	-mmpy-option=2	-Xmpy	except ARC HS	-Xmpy16 is implied
	wlh1, wlh2, wlh3, wlh	-Xmpy_cycles= 1,2,3,4,5	-mmpy-option={3-6}	-Xmpy_cycles=1, 2,3,4,5	ARC700, ARC EM	-Xmpy is implied if spec
	wlh7	-Xmac	-mmpy-option=7		ARC HS	-Xmac implies -Xmpy and
	wlh8	-Xmacd	-mmpy-option=8		ARC HS	-Xmacd implies -Xmac
	wlh9	-Xqmpyh	-mmpy-option=9		ARC HS	-Xqmpyh implies -Xmacd
	A600 MPY	-Xmult32		-Xmult32	ARC600	
	A600 MPY cycles	-Xmult32_cycles=N	-mulcost={1,2,3,4,5}	-Xmult32_cycles=N	ARC600	-Xmult32 is implied
64-bit Load and Store	ll64	-Xll64	-mll64	-Xll64	ARC HS	LDD/STD

Option	ARChitect	CCAC	GCC	MDB	Supported core	Notes
Unaligned Memory access		-XUnaligned	N.A.	-XUnaligned	ARC HS	Unaligned memory accesses
Atomic Option	atomic_option	-Xatomic	-matomic	-Xatomic		Default
Extended Arithmetic Instructions		-Xea	-mEA	-Xea	ARC600, ARC700	
Xlib MetaWare Opt		-Xlib	N.A	-Xlib	ARC600	Expansion: -Xmult32 -Xno
		-Xlib	N.A	-Xlib	ARC601	Expansion: -Xbs -Xmult32
		-Xlib	N.A.	-Xlib	ARC700	Expansion: -Xmpy (-Xnorm
		-Xlib	N.A.	-Xlib	ARC EM	Expansion: -Xbs -Xmpy -X
		-Xlib	N.A.	-Xlib	ARC HS	Expansion: -Xmpy -Xll64
Endianness	Big	-HB	-mbig-endian	N/A		MDB reads endianess from ELF
	Little	-HL	-mlittle-endian	N/A		
PC Width	pc_size	-Hpc_width= 16,20,24,28,	N.A.	-pc_width=16,20,24,28,32	except ARC HS	For HS, fixed pc_width=32
Loop Counter Width	lpc_size	-Hlpc_width= 0,8,12,16,2	N.A.	-lpc_width=8,12,16,20,24	except ARC HS	For HS, fixed lpc_width=32
Address size	addr_size	N/A	N.A.	-addr_size=16,20,24,32	ARC EM	For HS, fixed addr_size=32
Number of Interrupts	number_of_interrupts	N/A	N.A.	-interrupts=1..240	All	For 600,700 only 8,16,24
Interrupt Vector Base	invbase_preset	N/A	N.A.	-interrupt_base=addr	All	
Fast Interrupts	irq_option	N/A	N.A.	-firq	ARC EM, ARC HS	
External Interrupts	external_interrupts	N/A	N.A.	-ext_interrupts=1..240	ARC EM, ARC HS	

Option	ARChitect	CCAC	GCC	MDB	Supported core	Notes
Number of priority level	number_of_levels	N/A	N.A.	-interrupt_priorities=1..16	ARC EM, ARC HS	
Number of Registers (ngr_num_regs)	16		N.A.	N/A	all	MDB reads -rf16 info fro
	32	Default	Default	N/A	all	CCAC assumes 32 registers
Number of Register Bank	ngf_num_banks		N.A.	-rgf_num_banks=1,2,3,4	ARC EM, ARC HS	EM - 2 register banks, H
Number of banked registers	ngf_banked_regs	-Hrgf_banked_regs=N	N.A.	-rgf_banked_regs=4,8,16,32	ARC EM	For HS it is 32 (or 16 w
Actionpoints	num_actionpoints					
Timer 0		-Xtimer0	N.A.	-Xtimer0	All	
Timer 1		-Xtimer1	N.A.	-Xtimer1	All	
DCCM Size	dccm_size	N/A	N.A.	-dccm_size=size	All	User linker map file to
DCCM Base Address	dccm_base	N/A	N.A.	-dccm_base=addr	All	
ICCM Size	iccm0_size	N/A	N.A.	-iccm_size=size	All	
	iccm1_size			-iccm1_size=size		
ICCM Base Address	iccm0_base	N/A	N.A.	-iccm_base=addr, -iccm0_X	All	
	iccm1_base			-iccm1_base=addr	All	
Data Cache	dc_size, dc_bsize, c_ways,	N/A	-param l1-cache-size -param l1-cache-line-size N.A.	-dcache=csz,lsz,w,attrib csz is 512 to 32k lsz is 8,16,32,64,128,256 w is 1,2,4 attrib is a or o	All	csz=cache size, lsz=line size, w=ways, attrib is optional,random
Instruction Cache	ic_size ic_bsize ic_ways	N/A	N.A.	-icache=csz,lsz,w,attrib csz is 512 to 32k lsz is 8,16,32,64,128,256 w is 1,2,4 attrib is a or o	All	Same format as -dcache csz=cache size, lsz=line size, w=ways, attrib is optional,random

Option	ARChitect	CCAC	GCC	MDB	Supported core	Notes
Instruction Fetch Queue	ifqueue_size				ARC EM, ARC HS	
	ifqueue_burst_size				ARC EM, ARC HS	
DSP Dual 16x16 MAC		-Xxmac_d16		-Xxmac_d16	ARC600, ARC700	Adds instructions MULDW,
DSP 24x24 MAC		-Xxmac_24		-Xxmac_24	ARC600, ARC700	Adds instructions MULT,
DSP 32x16 MPY		-Xmul32x16	-mmul32x16	-Xmul32x16	ARC600, ARC700	Adds Instructions MULULW
DSP Dual Floating Point	32x16 MUL/MAC	-Xdmulpf		-Xdmulpf	ARC 600	Adds instructions DMULPF
DSP XY memory		-Xxy	-mxy	-Xxy	ARC600, ARC700	
	XY size	-Xxysize=size	N.A.	-Xxysize=size	ARC600, ARC700	
	XY banks	-Xxybanks=banks	N.A.	-Xxybanks=banks	ARC600, ARC700	
	XY base X	-Xxylsbasex=addr	N.A.	-Xxylsbasex=addr	ARC600, ARC700	
	XY base Y	-Xxylsbasey=addr	N.A.	-Xxylsbasey=addr	ARC600, ARC700	
FPX (floating pt.)	Single	-Xspfp	-mspfp	-Xspfp	except ARC HS	Also -Xspfp_compact, -Xs
	Double	-Xdpfp	-mdpfp	-Xdpfp	except ARC HS	Also -Xdpfp_compact, -Xd
eFPX (new fp unit)					ARC HS	No support in compiler/d

Option	ARChitect	CCAC	GCC	MDB	Supported core	Notes
SIMD Unit		-Xsimd	-msimd	-Xsimd	ARC700	
Time Stamp Counter		-Xrtsc	N.A.	-Xrtsc	ARC700	RTSC instruction
Stack Boundary Check		N/A	N.A.	-Xstack_check	ARC700	
Memory Management Unit	Version 2	N/A	N.A.	-mmu	ARC700	
	Version3	N/A	N.A.	-mmuv3 -prop=mmu_pagesize=NNN -prop=jtlb_ways=WWW -prop=mmu_osm=N	ARC700 ARC700 ARC700 ARC700	Provides more configurab Set the page size in byt change the number of way Set the OSM bit, N=1 or
FPU (IEEE fp)	Sngl		-mfpu=fpus		ARC EM, ARC HS	
	Sngl with DIV/SQRT		-mfpu=fpus_dis		ARC EM, ARC HS	
	Sngl with fused ops		-mfpu=fpus_fma		ARC EM, ARC HS	
	Sngl all extensions		-mfpu=fpus_all		ARC EM, ARC HS	
	Dbl		-mfpu=fpud		ARC HS	
	Dbl with DIV/SQRT		-mfpu=fpud_dis		ARC HS	
	Dbl with fused ops		-mfpu=fpud_fma		ARC HS	
	Dbl all extensions		-mfpu=fpud_all		ARC HS	
	Dbl assist insns		-mfpu=fpuda		ARC EM	

Migrating ARChitect APEX generated header to GNU

The ARChitect2 tool generates a special header file when APEX instructions are specified/selected in ARChitect. Such an example can be:

```
/* **** DO NOT EDIT - this file is generated by ARChitect2 ****
 *
 * Description: Header file declaring the compiler extensions for apex components
 */

#ifndef _apexextensions_H_
#define _apexextensions_H_

#define APEX_EXT_CORE_REGS_EXT_CORE_REGS_PRESENT        1

// User extension aux register auxreg0
#define AR_AUXREG0 0xfffff800
#pragma Aux_register(0xfffff800, name=>"auxreg0")

// User extension aux register auxreg1
#define AR_AUXREG1 0xfffff801
#pragma Aux_register(0xfffff801, name=>"auxreg1")

// User extension core register r32
#define CR_R32 32
#pragma Core_register(32, name=>"r32")

// User extension core register r33
#define CR_R33 33
#pragma Core_register(33, name=>"r33")

// User extension core register r34
#define CR_R34 34
#pragma Core_register(34, name=>"r34")

// User extension core register r35
#define CR_R35 35
#pragma Core_register(35, name=>"r35")

// User extension instruction insn1
extern long insn1(long,long);
#pragma intrinsic(insn1,opcode=>7,sub_opcode=>5, effects=>"reg=32:is_read:is_written; reg=33:is_r

// User extension instruction insn2
extern long insn2(long);
#pragma intrinsic(insn2,opcode=>7,sub_opcode=>1, effects=>"reg=32:is_read:is_written; reg=33:is_r

#endif
```

MWDT specific header analysis

Let us take each element of the autogenerated header and analyse it.

The Auxiliary registers

```
// User extension aux register auxreg0
#define AR_AUXREG0 0xfffff800
#pragma Aux_register(0xfffff800, name=>"auxreg0")
```

The MWDT compiler accepts the definition of various extension features via pragmas. However, this is not the case for GNU. In this case, we need to use inline assembly to make the toolchain aware of the added functionality. Thus, to handle auxiliary register definition at C-level, we may need to define:

```
#define Aux_register(ADDR, NAME)                \
  asm (".extAuxRegister " NAME ", " #ADDR ", r|w")
```

Having this instantiation:

```
Aux_register (0xfffff800, "auxreg0");
```

The extension core registers

```
// User extension core register r32
#define CR_R32 32
#pragma Core_register(32, name=>"r32")
```

Normally, GNU recognize all the extension core registers as r32-r57. Thus, it is not needed to define a core register which has the same name as in GNU.

User extension instructions

```
// User extension instruction insn1
extern long insn1(long,long);
#pragma intrinsic(insn1,opcode=>7,sub_opcode=>5, effects=>"reg=32:is_read:is_written; reg=33:is_r
```

MWDT format is using the function _insn1_ declaration to select the instruction syntax, while the pragma is defining the MOP, SOP and sides effects of the instruction. Thus, for our example, _insn1_ is using SYNTAX_3OP leading to:

```
#define intrinsic_3OP(NAME, MOP, SOP)           \
  asm (".extInstruction " NAME "," #MOP ","             \
  #SOP ",SUFFIX_NONE, SYNTAX_3OP\n\t")

intrinsic_3OP ("insn1", 7, 5);
```

for instruction declaration, and:

```
__extension__ static __inline int32_t __attribute__ ((__always_inline__))
insn1 (int32_t __a, int32_t __b)
{
  int32_t __dst;
  __asm__ ("insn1 %0, %1, %2\n\t"
           : "=r" (__dst)
           : "r" (__a), "rCal" (__b)
           : "r32", "r33", "r34", "r35");
  return __dst;
}
```

to be used in C.

The list of clobber registers can be ignored as the compiler does not handle the r32 to r56 register. However, to be safe, it is good to inform it about them. As for auxiliary register, one can ignore or just add “memory” to the clobber list as AUX registers are memory.

GNU header

```
#ifndef _apexextensions_H_
#define _apexextensions_H_

#include <stdint.h>

#define Aux_register(ADDR, NAME)                \
  asm (".extAuxRegister " NAME ", " #ADDR ", r|w")
```

```
#define intrinsic_3OP(NAME, MOP, SOP)           \
  asm (".extInstruction " NAME "," #MOP ","             \
  #SOP ",SUFFIX_NONE, SYNTAX_3OP\n\t")

#define intrinsic_2OP(NAME, MOP, SOP)           \
  asm (".extInstruction " NAME "," #MOP ","             \
  #SOP ",SUFFIX_NONE, SYNTAX_2OP\n\t")

Aux_register (0xfffff800, "auxreg0");
Aux_register (0xfffff801, "auxreg1");

intrinsic_3OP ("insn1", 7, 5);
intrinsic_2OP ("insn2", 7, 1);

__extension__ static __inline int32_t __attribute__ ((__always_inline__))
insn1 (int32_t __a, int32_t __b)
{
  int32_t __dst;
  __asm__ ("insn1 %0, %1, %2\n\t"
               : "=r" (__dst)
               : "r" (__a), "rCal" (__b)
               : "r32", "r33", "r34", "r35", "memory");
  return __dst;
}

__extension__ static __inline int32_t __attribute__ ((__always_inline__))
insn2 (int32_t __a)
{
  int32_t __dst;
  __asm__ ("insn2 %0, %1\n\t"
               : "=r" (__dst)
               : "rCal" (__a)
               : "r32", "r33", "r34", "r35", "memory");
  return __dst;
}

#endif
```

Intrinsics in ARC GNU vs MWDT

If one is interested in ARC-specific GCC built-ins those might be found in upstream documentation here: https://gcc.gnu.org/onlinedocs/gcc/ARC-Built-in-Functions.html.

Note to use listed below intrinsics it's required to include `arcle.h` in your source file that way:

```
#include <arcle.h>
```

List of ARC intrinsics supported by MetaWare & GCC compilers

MetaWare compiler	GCC compiler
_abss	_abss
_abssh	_abssh
_adcs	_adcs
_add	Not planned
_add1	Not planned
_add1_f	Not planned
_add2	Not planned

MetaWare compiler	GCC compiler
_add2_f	Not planned
_add3	Not planned
_add3_f	Not planned
_add_f	Not planned
_adds	_adds
_adds_f	Not planned
_aex	Unsupported
_and	Not planned
_and_f	Not planned
_asl	Not planned
_asl_f	Not planned
_aslacc	_aslacc
_asls	_asls
_asls_f	Not planned
_aslsacc	_aslsacc
_asr	Not planned
_asr_f	Not planned
_asrs	_asrs
_asrs_f	Not planned
_asrsr	_asrsr
_asrsr_f	Not planned
_bclr	Not planned
_bclr_f	Not planned
_bmsk	Not planned
_bmsk_f	Not planned
_bset	Not planned
_bset_f	Not planned
_btst_f	Not planned
_bxor	Not planned
_bxor_f	Not planned
_cbflyhf0r	_cbflyhf0r
_cbflyhf1r	_cbflyhf1r
_cmacchfr	_cmacchfr
_cmacchnfr	_cmacchnfr
_cmachfr	_cmachfr

MetaWare compiler	GCC compiler
_cmachnfr	_cmachnfr
_cmpychfr	_cmpychfr
_cmpychnfr	_cmpychnfr
_cmpyhfmr	_cmpyhfmr
_cmpyhfr	_cmpyhfr
_cmpyhnfr	_cmpyhnfr
_divf	_divf
_dmach	_dmach
_dmachbl	_dmachbl
_dmachbm	_dmachbm
_dmachf	_dmachf
_dmachfr	_dmachfr
_dmachu	_dmachu
_dmacwh	_dmacwh
_dmacwhf	_dmacwhf
_dmacwhu	_dmacwhu
_dmpyh	_dmpyh
_dmpyhbl	_dmpyhbl
_dmpyhbm	_dmpyhbm
_dmpyhf	_dmpyhf
_dmpyhfr	_dmpyhfr
_dmpyhu	_dmpyhu
_dmpyhwf	_dmpyhwf
_dmpywh	_dmpywh
_dmpywhf	_dmpywhf
_dmpywhu	_dmpywhu
_ex	unsupported
_ex_di	unsupported
_ffs	unsupported
_flagacc	_flagacc
_fls	unsupported
_getacc	_getacc
_kflag	_kflag
_lr	_lr
_lsr	Not planned

MetaWare compiler	GCC compiler
_lsr_f	Not planned
_mac	_mac
_macd	_macd
_macdf	_macdf
_macdu	_macdu
_macf	_macf
_macfr	_macfr
_macu	_macu
_macwhfl	_macwhfl
_macwhflr	_macwhflr
_macwhfm	_macwhfm
_macwhfmr	_macwhfmr
_macwhkl	_macwhkl
_macwhkul	_macwhkul
_macwhl	_macwhl
_macwhul	_macwhul
_max_f	Not planned
_min_f	Not planned
_modif	Not planned
_mov_f	Not planned
_mpy	Not planned
_mpyd	Not planned
_mpydf	_mpydf
_mpydu	Not planned
_mpyf	_mpyf
_mpyfr	_mpyfr
_mpym	Not planned
_mpymu	Not planned
_mpyu	Not planned
_mpywhfl	_mpywhfl
_mpywhflr	_mpywhflr
_mpywhfm	_mpywhfm
_mpywhfmr	_mpywhfmr
_mpywhkl	_mpywhkl
_mpywhkul	_mpywhkul

MetaWare compiler	GCC compiler
_mpywhl	_mpywhl
_mpywhul	_mpywhul
_msubdf	_msubdf
_msubf	_msubf
_msubfr	_msubfr
_msubwhfl	_msubwhfl
_msubwhflr	_msubwhflr
_msubwhfm	_msubwhfm
_msubwhfmr	_msubwhfmr
_negs	_negs
_negs_f	Not planned
_negsh	_negsh
_negsh_f	Not planned
_norm	Not planned
_norm_f	Not planned
_normacc	_normacc
_normh	Not planned
_normh_f	Not planned
_normw	Not planned
_normw_f	Not planned
_or	Not planned
_or_f	Not planned
_qmach	_qmach
_qmachf	_qmachf
_qmachu	_qmachu
_qmpyh	_qmpyh
_qmpyhf	_qmpyhf
_qmpyhu	_qmpyhu
_rndh	_rndh
_rndh_f	Not planned
_ror	Not planned
_ror_f	Not planned
_rrc	Not planned
_rrc_f	Not planned
_satf	_satf

MetaWare compiler	GCC compiler
_sath	_sath
_sath_f	Not planned
_sbcs	_sbcs
_setacc	_setacc
_sqrt	_sqrt
_sqrtf	_sqrtf
_sr	_sr
_sub	Not planned
_sub1	Not planned
_sub1_f	Not planned
_sub2	Not planned
_sub2_f	Not planned
_sub3	Not planned
_sub3_f	Not planned
_sub_f	Not planned
_subs	_subs
_subs_f	Not planned
_trap	_trap
_vabs2h	_vabs2h
_vabss2h	_vabss2h
_vadd2	_vadd2
_vadd2h	_vadd2h
_vadd4b	_vadd4b
_vadd4h	_vadd4h
_vadds2	_vadds2
_vadds2h	_vadds2h
_vadds4h	_vadds4h
_vaddsub	_vaddsub
_vaddsub2h	_vaddsub2h
_vaddsub4h	_vaddsub4h
_vaddsubs	_vaddsubs
_vaddsubs2h	_vaddsubs2h
_vaddsubs4h	_vaddsubs4h
_valgn2h	_valgn2h
_vasl2h	_vasl2h

MetaWare compiler	GCC compiler
_vasls2h	_vasls2h
_vasr2h	_vasr2h
_vasrs2h	_vasrs2h
_vasrsr2h	_vasrsr2h
_vext2bhl	_vext2bhl
_vext2bhlf	_vext2bhlf
_vext2bhm	_vext2bhm
_vext2bhmf	_vext2bhmf
_vlsr2h	_vlsr2h
_vmac2h	_vmac2h
_vmac2hf	_vmac2hf
_vmac2hfr	_vmac2hfr
_vmac2hnfr	_vmac2hnfr
_vmac2hu	_vmac2hu
_vmax2h	_vmax2h
_vmin2h	_vmin2h
_vmpy2h	_vmpy2h
_vmpy2hf	_vmpy2hf
_vmpy2hfr	_vmpy2hfr
_vmpy2hu	_vmpy2hu
_vmpy2hwf	_vmpy2hwf
_vmsub2hf	_vmsub2hf
_vmsub2hfr	_vmsub2hfr
_vmsub2hnfr	_vmsub2hnfr
_vneg2h	_vneg2h
_vnegs2h	_vnegs2h
_vnorm2h	_vnorm2h
_vpack2hbl	_vpack2hbl
_vpack2hblf	_vpack2hblf
_vpack2hbm	_vpack2hbm
_vpack2hbmf	_vpack2hbmf
_vpack2hl	_vpack2hl
_vpack2hm	_vpack2hm
_vperm	_vperm
_vrep2hl	_vrep2hl

MetaWare compiler	GCC compiler
_vrep2hm	_vrep2hm
_vsext2bhl	_vsext2bhl
_vsext2bhm	_vsext2bhm
_vsub2	_vsub2
_vsub2h	_vsub2h
_vsub4b	_vsub4b
_vsub4h	_vsub4h
_vsubadd	_vsubadd
_vsubadd2h	_vsubadd2h
_vsubadd4h	_vsubadd4h
_vsubadds	_vsubadds
_vsubadds2h	_vsubadds2h
_vsubadds4h	_vsubadds4h
_vsubs2	_vsubs2
_vsubs2h	_vsubs2h
_vsubs4h	_vsubs4h
_wevt	Unsupported

Improving GCC output

Understanding compiler options

There are cases when using solely the -Ox options will not bring the desired optimization (either size or speed) for a compiled function/application. In these cases we need to understand where is the program's bottleneck and if it can be solved either by passing various options to the compiler or by source code modifications. In this section, we look into compiler's command-line options and how they can help us in achieving better results.

Architecture-Independent Optimizations

The first step in optimizing your code is by experimenting with architecture-independent optimizations. Almost any GCC pass (i.e., optimization) can be turned on or off or steered using parameters. These optimizations are denoted by the notation -f*xxxx*, where *xxxx* is the GCC pass that is turned on. To turn off a gcc pass, we need to pass -fno-*xxxx* to the compiler. The same observation holds for other types of optimizations such as the architecture-specific ones. For more information about GCC options, please check the GCC manual. It is highly desirable to know and understand how these options work in order to properly use them.

To avoid being overwhelmed by the sheer amount of options available, I use for my day-to-day source code exploration the following tree related options (either on or off):

`-ftree-loop-ivcanon`

Create a canonical counter for number of iterations in loops for which determining number of iterations requires complicated analysis. Later optimizations then may determine the number easily. Useful especially in connection with unrolling.

`-ftree-vectorize`

Perform loop vectorization on trees. This flag is enabled by default at -O3. This option is useful to use either if the ARC processor doesn't have the SIMD extensions as it performs extra code analysis and may improve the following optimizations.

`-ftree-loop-if-convert`

Attempt to transform conditional jumps in the innermost loops to branch-less equivalents. The intent is to remove control-flow from the innermost loops in order to improve the ability of the vectorization pass to handle these loops. This is enabled by default if vectorization is enabled.

`-f[no-]tree-dominator-opts`

Perform a variety of simple scalar cleanups (constant/copy propagation, redundancy elimination, range propagation and expression simplification) based on a dominator tree traversal. This also performs jump threading (to reduce jumps to jumps). This flag is enabled by default at -O and higher.

`-f(no-)ivopts`

Perform induction variable optimizations (strength reduction, induction variable merging and induction variable elimination) on trees. Disabling the `ivopts` optimization may improve the number of hardware loops recognized by the compiler.

`-fselective-scheduling`

Schedule instructions using selective scheduling algorithm. Selective scheduling runs instead of the first scheduler pass.

`-fgcse`

Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation. It may be useful to disable this step specially when we want to have more SUB1/2/3, ADD1/2/3 type of operations generated.

`-frename-registers`

Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization most benefits processors with lots of registers. Depending on the debug information format adopted by the target, however, it can make debugging impossible, since variables no longer stay in a "home register". Enabled by default with `-funroll-loops` and `-fpeel-loops`.

`-fira-loop-pressure`

Use IRA to evaluate register pressure in loops for decisions to move loop invariants. This option usually results in generation of faster and smaller code on machines with large register files (>= 32 registers), but it can slow the compiler down.

`-fsched-pressure`

Enable register pressure sensitive insn scheduling before register allocation. This only makes sense when scheduling before register allocation is enabled, i.e. with `-fschedule-insns`. Usage of this option can improve the generated code and decrease its size by preventing register pressure increase above the number of available hard registers and subsequent spills in register allocation.

`-f[no-]regmove`

Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. This is especially helpful on machines with two-operand instructions. Disabling this optimization may result in faster code.

Processor-Specific Optimizations

ARC GCC specific backend switches can be used to improve the code size or code speed. We need always to use the ARC switches that enables usage of the hardware extensions (such as -mdiv-rem). An overview of those options can be found in ARC's gcc manual or by invoking gcc with `--help=target`. Additionally, I use the next switches to enable better handling of LD/ST operations:

`-mindexed-loads`

Enable the use of indexed loads. This can be problematic because some optimizers will then assume that indexed stores exist, which is not the case.

`-mauto-modify-reg`

Enable the use of pre/post modify with register displacement.

GCC optimizations for Code Size

If code size is our target, beside the GCC's -Os option, it may make sense to use it in conjunction with following command-line options:

- `-fsection-anchors`
- `-fno-branch-count-reg`
- `-fira-loop-pressure`
- `-fira-region=all`
- `-fno-sched-spec-insn-heuristic`
- `-fno-move-loop-invariants`
- `-fno-tree-dominator-opts`
- `-ftree-vectorize`
- `-fno-cse-follow-jumps`
- `-fno-jump-tables`

I would advise compiling a program with -O2 and -Os and comparing runtime performance and memory footprint. It may be that the code is as fast as compiled with -O2 but smaller due to -Os option.

GCC optimization for speed

If the cycle count is our target, the best is to start with -O2 option then with -O3 and for each compiler optimization level to combine one or more of the suggested GCC command-line options. Finally, gather and compare runtime performance and size for each command-line combination. I suggest to plot these numbers on a 2-D graph, where one axis will represent the cycle count, and the other will represent the size. Hence, we can choose the best combination size/speed for a given problem.

If one wants to try a large number of option combinations, then an automatic scripting process is required. One of those tools that searches through more than 1.3 zillion gcc option combination is Acovea. Acovea is using genetic algorithms to search for the best option combination for a given program. However, one can make an script that uses only the suggested gcc options to search for the best combination by exhaustively generating (most) of the option combinations.

Using _optimize_ attribute

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully. In the case when we want a certain function/kernel not to change its speed/size characteristics, we can use the _optimize_ function attribute. The _optimize_ attribute is used to specify that a function is to be compiled with different optimization options than specified on the command line. Arguments can either be numbers or strings. Numbers are assumed to be an optimization level. Strings that begin with O are assumed to be an optimization option, while other options are assumed to be used with a -f prefix.

Default GCC driver options and parameters; ARC specific

Optimizations

Optimizations	O0	O1	Os	O2	O3
fomit-frame-pointer		On	On	On	On
fschedule-insns		On	On	On	On
fschedule-insns2		On	On	On	On
mearly-cbranchsi	On	On	On	On	On
mbbit-peephole	On	On	On	On	On
mcase-vector-pcrel			On		
mcompact-casesi			On		

Parameters

Parameter	Value
simultaneous-prefetches	4
prefetch-latency	4
l1-cache-line-size	64

ARC hardware variation

CPU	mpy	barrel shifter	norm	swap	atomic	mpy16	code density	divrem	ll64
ARC600	N.A.	On	Off	Off	N.A.	N.A.	N.A.	N.A.	N.A.
ARC601	N.A.	Off	Off	Off	N.A.	N.A.	N.A.	N.A.	N.A.
ARC700	On	On	On	Off	Off	N.A.	N.A.	N.A.	N.A.
ARC EM	On	On	Off	Off	Off	On	Off	Off	N.A.
ARC HS	On	On	On	On	On	On	On	On	On

Information for Toolchain maintainers

Creating toolchain release

Introduction

`release.mk` is a makefile to create prebuilt packages of GNU Toolchain for ARC. It relies on other scripts in toolchain repository to build components.

To build a release GNU Toolchain for ARC processors, several prerequisites should be installed and/or built before running a `release.mk` which does most of the work. List of prerequisites to build toolchain is list in toolchain `README.md` file. Note that there are extra dependencies to build toolchain for Windows hosts, in addition to those that are required to build toolchain for Linux hosts. To create Windows installer several MinGW and MSYS

components are required (path set by THIRD_PARTY_SOFTWARE_LOCATION). For a list of MinGW and MSYS packages, please refer to *windows-installer/README.md* section “Prerequisites”.

There are several variables that can be set to disable particular components, like Windows installer or OpenOCD, however those are not specifically tested, so may not really work. By default `release.mk` will build all of the possible components. It is also possible to invoke particular Make targets directly to get only a limited set of distributables, however it is not possible to make further targets like deploy or upload to use only this limited set of files (there is always an option to modify `release.mk` to get desired results).

Building prerequisites

Eclipse plugin for ARC

> **Warning**
>
> This section doesn’t cover a `build/Makefile` file in `arc_gnu_eclipse` repository which automates build process with ant.

Build Eclipse plugin for ARC following those guidelines: https://github.com/foss-for-synopsys-dwc-arc-processors/arc_gnu_eclipse/wiki/Creating-Eclipse-plugins-release-zip-file Create and push respective git tag:

```
$ pushd arc_gnu_eclipse
$ git tag arc-2016.03
$ git push -u origin arc-2016.03
$ popd
```

Environment variables

Those are make variables which can be set either as a parameters to make, like `make PARAM=VALUE` or they can be specified in the `release.config` file that will be sourced by `release.mk`.

CONFIG_STATIC_TOOLCHAIN
: Whether to build toolchain linked dynamically or statically. Note this affects the toolchain executable files, not the target libraries.

  Possible values

  y and n

  Default value

  n

DEPLOY_BUILD_DESTINATION
: Where to copy unpacked engineering build. Location is in format `[hostname:]/path`. A directory named `${RELEASE_TAG##-arc}` will be created in the target path and will contain unpacked directories. Directories names are different from those that are in the tarballs - namely useless cruft is avoided and verion is not mentioned as well, so that it is easier to use those directories via symbolic links. For example, for tarball arc_gnu_2016.09-eng006_prebuilt_elf32_le_linux_install.tar.gz build directory will be elf32_le_linux.

DEPLOY_DESTINATION
: Where to copy release distributables. Location is in format `[hostname:]/path`. A directory named `${RELEASE_TAG##-arc}` will be created in the target path and will contain all deploy artifacts. So for `RELEASE_TAG = arc-2016.03-alpha1` directory will be `2016.03-alpha1`, while for `RELEASE_TAG = arc-2016.03` it will be `2016.03`.

ENABLE_BIG_ENDIAN
: Whether to build and upload big endian toolchain builds. Big endian toolchain is required for IDE targets.

Possible values

y and n

Default value

y

ENABLE_DOCS_PACKAGE

Whether to build separate packages with just documentation PDF files.

Possible values

y and n

Default value

n

ENABLE_IDE

Whether to build and upload IDE distributable package. Note that build script for Windows installer always assumes presence of IDE, therefore it is not possible to build it when this option is n.

Possible values

y and n

Default value

y

ENABLE_IDE_MACOS

Whether to build and upload IDE distributable package on macOS.

Possible values

y and n

Default value

n

ENABLE_IDE_PLUGINS_BUILD

Whether to build IDE plugins as part of ARC GNU Toolchain or download prebuilt ZIP.

Possible values

y and n

Default value

y

ENABLE_LINUX_IMAGES

Whether to build and deploy Linux images built with this toolchain. This targets uses Buildroot to build rootfs and uImage for AXS103.

Possible values

y and n

Default value

y

ENABLE_LINUX_TOOLS

Whether to build and deploy GNU Toolchain for Linux targets.

Possible values

y and n

Default value

y

ENABLE_GLIBC_TOOLS

Whether to build and deploy GNU Toolchain for Linux/glibc targets.

Possible values

y and n

Default value

y

ENABLE_NATIVE_TOOLS

Whether to build and upload native toolchain. Currently toolchain is built only for ARC HS Linux.

Possible values

y and n

Default value

y

ENABLE_OPENOCD

Whether to build and upload OpenOCD distributable package for Linux. IDE targets will not work if OpenOCD is disabled. Therefore if this is n, then :envvar:`ENABLE_IDE` and ENABLE_WINDOWS_INSTALLER` also must be n.

Possible values:

y and n

Default value:

y

ENABLE_OPENOCD_WIN

Whether to build and upload OpenOCD for Windows. This target currently depends on ENABLE_OPENOCD, which causes source code to be cloned for OpenOCD. OpenOCD for Windows build will download and build libusb library and is a prerequisite for IDE for Windows build.

Possible values

y and n

Default value

y

ENABLE_PDF_DOCS

Whether to build Toolchain PDF documentation. This affects only the “toolchain” repository - PDF documents from gcc, binutils, etc are always created, regardless of this option.

Possible values

y and n

Default value

y

ENABLE_SOURCE_TARBALL

Whether to create a source tarball. Usually that should be true, so that release would include source tarball, however if release makefile is run multiple times on various machines to create packages for various target systems, then it makes sense to have this true only on one system.

Possible values

y and n

Default value

y

ENABLE_WINDOWS_INSTALLER

Whether to build and upload Windows installer for toolchain and IDE. While building of installer can be also skipped simply by not invoking respective make targets, installer files still will be in the list of files that should be deployed and uploaded to GitHub, therefore this variable should be set to n for installer to be completely skipped. This variable also disables build of the toolchain for Windows as well.

Possible values

y and n

Default value

y

GIT_REFERENCE_ROOT
: Root location of existing source tree with all toolchain components Git repositories. Those repositorie swill be used as a reference when cloning source tree - this reduces time to clone and disk space consumed. Note that all of the components must exist in reference root, otherwise clone will fail.

IDE_PLUGIN_LOCATION
: Location of ARC plugin for Eclipse. This must be a directory and plugin file must have a name `arc_gnu_${RELEASE_TAG##arc-}_ide_plugin.zip`. File will be copied with rsync therefore location may be prefixed with hostname separated by semicolon, as in `host:/path`. This variable is used and must be set only if :envvar:`ENABLE_IDE_PLUGINS_BUILD` is set to `n`.

LIBUSB_VERSION
: Version of Libusb used for OpenOCD build for Windows.

  Default value

  1.0.20

RELEASE_NAME
: Name of the release, for example “GNU Toolchain for ARC Processors, 2016.03”.

RELEASE_TAG
: Git tag for this release. Tag is used literally and can be for example, arc-2016.03-alpha1.

THIRD_PARTY_SOFTWARE_LOCATION
: Location of 3rd party software, namely Java Runtime Environment (JRE) and Eclipse tarballs.

WINDOWS_TRIPLET
: Triplet of MinGW toolchain to do a cross-build of toolchain for Windows.

  Default value

  i686-w64-mingw32

WINDOWS_WORKSPACE
: Path to a directory that is present on build host and is also somehow available on a Windows host where Windows installer will be built. Basic scenario is when this location is on the Linux hosts, shared via Samba/CIFS and mounted on Windows host. Note that on Windows path to this directory, should be as short as possible , because Eclipse contains very long file names, while old NSIS uses ancient Windows APIs, which are pretty limited in the maximum file length. As a result build might fail due to too long path, if WINDOWS_LOCATION is too long on Windows host.

Make targets

build
: Build all distributable components that can be built on RHEL hosts. The only components that are not built by this target are:

  - OpenOCD for Windows - (has to be built on Ubuntu
  - ARC plugins for Eclipse - built by external job
  - Windows installer - created on Windows hosts. This tasks would depend on toolchain created by build target.

  This target is affected by RELEASE_TAG.

copy-windows-installer
: Copy Windows installer, created by `windows-installer/build-installer.sh` from WINDOWS_WORKSPACE to `release_output` directory.

create-tag
: Create Git tags for released components. Required environment variables: RELEASE_TAG, RELEASE_NAME. OpenOCD must have a branch named `arc-0.9-dev-${RELEASE_BRANCH}`, where `RELEASE_BRANCH` is a bare release, evaluated from the tag, so for RELEASE_TAG of `arc-2016.09-eng003`, `RELEASE_BRANCH` would be `2016.09`.

deploy
: Deploy build artifacts to remote locations. It deploys same files as those that are released, and a few extra ones (like Windows toolchain tarballs). This target just copies deploy artifacts to location specified by

DEPLOY_DESTINATION. This target depends on DEPLOY_DESTINATION and on WINDOWS_WORKSPACE.

distclean
: Remove all cloned sources as well as build artifacts.

prerequisites
: Clone sources of toolchain components from GitHub. Copy external components from specified locations. Is affected by following environment variables: RELEASE_TAG, GIT_REFERENCE_ROOT (optional), THIRD_PARTY_SOFTWARE_LOCATION.

push-tag
: Push Git tags to GitHub.

upload
: Upload release distributables to GitHub Releases. A new GitHub "Release" is created and bound to the Git tag specified in RELEASE_TAG. This target also depends on RELEASE_NAME to specify name of release on GitHub.

windows-workspace
: Create a workspace to run `windows-installer/build-installer.sh` script. Location of workspace is specified with WINDOWS_WORKSPACE. `build-installer.sh` script will create an installer in the workspace directory. To copy installer from workspace to `release_output` use copy-windows-installer.

Invocation

Release process consists of several sequential steps that should be done in the specified order. Some custom modifications can be done in between those steps.

First, create directory-workspace:

```
$ mkdir arc-2016.03
$ cd arc-2016.03
```

Clone the `toolchain` repository:

```
$ git clone -b arc-dev \
  https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain.git
```

That command uses an HTTPS protocol to do Git clone - other protocols may be used as well. This documentation assumes the default case where `arc-dev` branch is the base for the release.

Note

Currently `tag-release.sh` script used in the release process has a check that ensures that current branch is a development branch by checking that branch name ends in `-dev`.

First setup required make variables in the `release.config` file that will be sourced by `release.mk` (`...` must be replaced with an actual paths):

```
$ cat release.config
RELEASE_TAG=arc-2016.03
THIRD_PARTY_SOFTWARE_LOCATION=...
GIT_REFERENCE_ROOT=...
WINDOWS_WORKSPACE=...
```

Fetch prerequisites (git repositories and external packages):

```
$ make -f release.mk prerequisites
```

Create git tags:

```
$ make -f release.mk create-tag
```

Build toolchain:

```
$ make -f release.mk build
```

Prepare workspace for Windows installer build script. Note that target location, as specified by WINDOWS_WORKSPACE should be shared with Windows host on which installer will be built.

```
$ make -f release.mk windows-workspace
```

On Windows host, build installer using `windows-installer/build-installer.sh` script. Note that this script requires a basic cygwin environment.

```
$ RELEASE_BRANCH=2016.03 toolchain/windows-installer/build-installer.sh
```

Copy Windows installer from WINDOWS_WORKSPACE into `release_output`:

```
$ make -f release.mk copy-windows-installer
```

Deploy toolchain to required locations. This target may be called multiple times with different DEPLOY_DESTINATION values:

```
$ make -f release.mk deploy DEPLOY_DESTINATION=<site1:/pathA>
$ make -f release.mk deploy DEPLOY_DESTINATION=<site2:/pathB>
```

Similarly, unpacked builds can be deployed to multiple locations:

```
$ make -f release.mk deploy-build DEPLOY_BUILD_DESTINATION=<site1:/pathC>
$ make -f release.mk deploy-build DEPLOY_BUILD_DESTINATION=<site2:/pathD>
```

Push tags to remote repositories:

```
$ make -f release.mk push-tag
```

Finally, upload assets to GitHub Releases:

```
$ make -f release.mk upload
```

How to Build GNU Toolchain for ARC Manually

This document is a quick set of commands to build GNU toolchain for ARC manually, without **build-all.sh** script from this repository. Those instructions do everything mostly the same, sans scripting sugar. In general it is recommended to build GNU Toolchain for ARC using the build-all.sh script. This document describes what is done by this scripts and can be useful for situations where those scripts do not work for one or another reason.

It is assumed that current directory is top level directory, which contains all of the requires git repositories checked out.

Baremetal (elf32) toolchain

Build GNU binutils:

```
$ mkdir -p build/binutils
$ cd build/binutils
$ ../../binutils/configure \
    --target=arc-elf32|arceb-elf32 \
    --with-cpu=arcem|archs|arc700|arc600 \
    --disable-multilib|--enable-multilib \
    --enable-fast-install=N/A \
    --with-endian=little|big \
    --disable-werror \
    --enable-languages=c,c++ \
    --with-headers=../../newlib/newlib/libc/include \
    --prefix=${INSTALLDIR}
$ make {all,pdf}-{binutils,gas,ld}
$ make install-{,pdf-}{binutils,gas,ld}
$ cd -
```

Build GCC, but without libstdc++. Libstdc++ requires libc which is not available at that stage:

```
$ mkdir -p build/gcc
$ cd build/gcc
$ ../../gcc/configure \
    --target=arc-elf32|arceb-elf32 \
    --with-cpu=arcem|archs|arc700|arc600 \
    --disable-multilib|--enable-multilib \
    --enable-fast-install=N/A \
    --with-endian=little|big \
    --disable-werror \
    --enable-languages=c,c++ \
    --with-headers=../../newlib/newlib/libc/include \
    --prefix=${INSTALLDIR}
$ make all-{gcc,target-libgcc} pdf-gcc
$ make install-{gcc,-target-libgcc,pdf-gcc}
$ cd -
```

Build newlib, build tools should be added to the PATH:

```
$ export PATH=$INSTALLDIR:$PATH
$ mkdir -p build/newlib
$ cd build/newlib
$ ../../newlib/configure \
    --target=arc-elf32|arceb-elf32 \
    --with-cpu=arcem|archs|arc700|arc600 \
    --disable-multilib|--enable-multilib \
    --enable-fast-install=N/A \
    --with-endian=little|big \
    --disable-werror \
    --enable-languages=c,c++ \
    --with-headers=../../newlib/newlib/libc/include \
    --prefix=${INSTALLDIR}
$ make all-target-newlib
$ make install-target-newlib
$ cd -
```

Now it is possible to build listdc++. Note extra options passed to configure. Without –disable-gcc and –disable-libgcc make would try to build those two once again, and without –with-newlib configuration will fail with “unsupported target/host combination”. Another option is to build libstdc++ using build tree of GCC and calling `make all-target-libstdc++-v3`, in that case there is no need to call “configure” separately, but –with-newlib should be passed when configuring GCC. Note that in case of a separate build directory things might

get awry if there is already a previous version of toolchain at the installation target location and in that case it is required to use build tree of GCC. Command to build listdc++:

```
$ mkdir -p build/libstdc++-v3
$ cd build/libstdc++-v3
$ ../../gcc/configure \
    --target=arc-elf32|arceb-elf32 \
    --with-cpu=arcem|archs|arc700|arc600 \
    --disable-multilib|--enable-multilib \
    --enable-fast-install=N/A \
    --with-endian=little|big \
    --disable-werror \
    --enable-languages=c,c++ \
    --with-headers=../../newlib/newlib/libc/include \
    --prefix=${INSTALLDIR} \
    --disable-gcc --disable-libgcc --with-newlib
$ make all-target-libstdc++-v3
$ make install-target-libstdc++-v3
$ cd -
```

Finally build GDB. GDB is the only component here that can be built in any order, as it doesn't depend on other components:

```
$ mkdir -p build/gdb
$ cd build/gdb
$ ../../gdb/configure \
    --target=arc-elf32|arceb-elf32 \
    --with-cpu=arcem|archs|arc700|arc600 \
    --disable-multilib|--enable-multilib \
    --enable-fast-install=N/A \
    --with-endian=little|big \
    --disable-werror \
    --enable-languages=c,c++ \
    --with-headers=../../newlib/newlib/libc/include \
    --prefix=${INSTALLDIR}
$ make {all,pdf}-gdb
$ make install-{,pdf-}gdb
$ cd -
```

Linux toolchain

uClibc toolchain

Define location of sysroot directory:

```
$ export SYSROOTDIR=$INSTALLDIR/arc-snps-linux-uclibc/sysroot
```

Install Linux headers:

```
$ cd linux
$ make ARCH=arc defconfig
$ make ARCH=arc INSTALL_HDR_PATH=$SYSROOTDIR/usr headers_install
$ cd -
```

Build binutils:

```
$ mkdir -p build/binutils
$ cd build/binutils
$ ../../binutils/configure \
    --target=arc-snps-linux-uclibc \
```

```
    --with-cpu=archs \
    --enable-fast-install=N/A \
    --with-endian=little \
    --disable-werror \
    --enable-languages=c,c++ \
    --prefix=${INSTALLDIR} \
    --enable-shared \
    --without-newlib \
    --disable-libgomp \
    --with-sysroot=$SYSROOTDIR
$ make all-{binutils,gas,ld}
$ make install-{binutils,ld,gas}
$ cd -
```

Build Stage 1 GCC (without libgcc):

```
$ mkdir -p build/gcc-stage1
$ cd build/gcc-stage1
$ ../../gcc/configure \
    --target=arc-snps-linux-uclibc \
    --with-cpu=archs \
    --disable-fast-install \
    --with-endian=little \
    --disable-werror \
    --disable-multilib \
    --enable-languages=c \
    --prefix=${INSTALLDIR} \
    --without-headers \
    --enable-shared \
    --disable-libssp \
    --disable-libmudflap \
    --without-newlib \
    --disable-c99 \
    --disable-libgomp \
    --with-sysroot=$SYSROOTDIR
$ make all-gcc
$ make install-gcc
$ cd -
```

Install uClibc headers:

```
$ cd uClibc
$ make ARCH=arc arcv2_defconfig
$ sed \
    -e "s#%KERNEL_HEADERS%#$SYSROOTDIR/usr/include#" \
    -e "s#%RUNTIME_PREFIX%#/#" \
    -e "s#%DEVEL_PREFIX%#/usr/#" \
    -e "s#CROSS_COMPILER_PREFIX=\".*\"#CROSS_COMPILER_PREFIX=\"arc-snps-linux-uclibc-\"#" \
    -i .config
$ make ARCH=arc PREFIX=$SYSROOTDIR install_headers
$ cd -
```

Build libgcc using build tree of stage 1 GCC:

```
$ cd build/gcc-stage1
$ make all-target-libgcc
$ make install-target-libgcc
$ cd -
```

Build uClibc:

```
$ cd uClibc
$ make ARCH=arc PREFIX=$SYSROOTDIR
$ make ARCH=arc PREFIX=$SYSROOTDIR install
$ cd -
```

Build Stage 2 GCC:

```
$ mkdir -p build/gcc-stage2
$ cd build/gcc-stage2
$ ../../gcc/configure \
    --target=arc-snps-linux-uclibc \
    --with-cpu=archs \
    --enable-fast-install=N/A \
    --with-endian=little \
    --disable-werror \
    --enable-languages=c,c++ \
    --prefix=${INSTALLDIR} \
    --enable-shared \
    --without-newlib \
    --disable-libgomp \
    --with-sysroot=$SYSROOTDIR
$ make all-{gcc,target-libgcc,target-libstdc++-v3}
$ make install-{gcc,target-libgcc,target-libstdc++-v3}
$ cd -
```

Build GDB:

```
$ mkdir -p build/gdb
$ cd build/gdb
$ ../../gcc/configure \
    --target=arc-snps-linux-uclibc \
    --with-cpu=archs \
    --enable-fast-install=N/A \
    --with-endian=little \
    --disable-werror \
    --enable-languages=c,c++ \
    --prefix=${INSTALLDIR} \
    --enable-shared \
    --without-newlib \
    --disable-libgomp \
    --with-sysroot=$SYSROOTDIR
$ make all-gdb
$ make install-gdb
$ cd -
```

Glibc toolchain

Glibc toolchain is built like the uClibc toolchain, but there are few differences. First, it is needed to change `--target=arc-snps-linux-uclibc` to `--target=arc-snps-linux-gnu`. Second, uClibc-specific stages should be replaced with following glibc-specific stages.

Install glibc headers:

```
$ mkdir -p build/glibc
$ cd build/glibc
$ ../../glibc/configure \
    --target=arc-snps-linux-gnu \
    --build=x86_64-pc-linux-gnu \
    --host=arc-snps-linux-gnu \
    --with-headers=$SYSROOTDIR/usr/include \
    --prefix=/usr \
    --disable-werror \
    --enable-obsolete-rpc
```

```
$ make install-bootstrap-headers=yes install-headers DESTDIR=$SYSROOTDIR
$ touch $SYSROOTDIR/usr/include/gnu/stubs.h
$ cd -
```

Build glibc:

```
$ cd build/glibc
$ make
$ make install DESTDIR=$SYSROOTDIR
$ cd -
```

Frequently asked questions

Compiling

- **Q: How to change heap and stack size in baremetal applications?**

  A: To change size of heap in baremetal applications the following option should be specified to the linker: `--defsym=__DEFAULT_HEAP_SIZE=${SIZE}`, where `${SIZE}` is desired heap size, in bytes. It also possible to use size suffixes, like `k` and `m` to specify size in kilobytes and megabytes respectively. For stack size respective option is `--defsym=__DEFAULT_STACK_SIZE=${STACK_SIZE}`. Note that those are linker commands - they are valid only when passed to “ld” application, if gcc driver is used for linking, then those options should be prefixed with `-Wl`. For example:

  ```
  $ arc-elf32-gcc -Wl,--defsym=__DEFAULT_HEAP_SIZE=256m \
    -Wl,--defsym=__DEFAULT_STACK_SIZE=1024m --specs=nosys.specs \
    hello.o -o hello.bin
  ```

  Those options are valid only when default linker script is used. If custom linker script is used, then effective way to change stack/heap size depends on properties of that linker script - it might be the same, or it might be different.

- **Q: Linker fails with error: ``undefined reference to `_exit’``. Among other possible functions are also _sbrk, _write, _close, _lseek, _read, _fstat, _isatty.**

  A: Function `_exit` is not provided by the libc itself, but must be provided by the libgloss, which is basically a BSP (board support package). Currently two libgloss implementations are provided for ARC: generic libnosys and libnsim which implements nSIM IO hostlink. In general libnosys is more suitable for hardware targets that doesn’t have hostlink support, however libnsim has a distinct advantage that on exit from application and in case of many errors it will halt the core, while libnosys will cause it to infinitely loop on one place. To use libnsim, pass option `--specs=nsim.specs` to gcc at link stage. If you are a chip or board developer, then it is likely that you would want to implement libgloss specific to your hardware.

- **Q: I’ve opened hs38.tcf and gcc options include ``-mcpu=hs34``. Why hs34 instead of hs38?**

  A: Possible values of `-mcpu=` options are orthogonal to names of IPlib templates and respective TCF. GCC option `-mcpu=` supports both `hs34` and `hs38` values, but they are different - `hs38` enables more features, like `-mll64` which are not present in `hs34`. ARC HS IPlib template hs38 doesn’t contain double-word load/store, therefore `-mcpu=hs38` is not compatible with this template. `-mcpu=hs34`, however, is compatible and that is why TCF generator uses this value. See Understanding GCC -mcpu option for a full list of possible `-mcpu` values and what IPlibrary templates they correspond to.

Debugging

- **Q: There are ``can’t resolve symbol`` error messages when using gdbserver on Linux for ARC target**

A: This error message might appear when gdbserver is a statically linked application. Even though it is linked statically, gdbserver still opens `libthread_db.so` library using `dlopen()` function. There is a circular dependency here, as `libthread_db.so` expects several dynamic symbols to be already defined in the loading application (gdbserver in this case). However statically linked gdbserver doesn't export those dynamic symbols, therefore `dlopen()` invocation causes those error messages. In practice there haven't been noticed any downside of this, even when debugging applications with threads, however that was tried only with simple test cases. To fix this issue, either rebuild gdbserver as a dynamically linked application, or pass option `--with-libthread-db=-lthread_db` to `configure` script of script. In this case gdbserver will link with `libthread_db` statically, instead of opening it with `dlopen()` and dependency on symbols will be resolved at link time.

- **Q: GDB prints an error message that ``XML support has been disabled at compile time``.**

  A: GDB uses Expat library to parse XML files. Support of XML files is optional for GDB, therefore it can be built without Expat available, however for ARC it usually required to have support of XML to read target description files. Mentioned error message might happen if GDB has been built without available development files for the Expat. On Linux systems those should be available as package in package manager. If Expat development files are not available for some reason, then pass option `--no-system-expat` to `build-all.sh` - with this option script will download and build Expat on it's own. That is especially useful when cross compiling for Windows hosts using Mingw, if development files of Expat are not available in the used Mingw installation.

ARC Development Systems

- **Q: How to reset ARC SDP board programmatically (without pressing "Reset" button)?**

  A: It is possible to reset ARC SDP board without touching the physical button on the board. This can be done using the special OpenOCD openocd:

  ```
  $ openocd -f test/arc/reset_sdp.tcl
  ```

  Note that OpenOCD will crash with a segmentation fault after executing this script - this is expected and happens only after board has been reset, but that means that other OpenOCD scripts cannot be used in chain with `reset_sdp.tcl`, first OpenOCD should be invoked to reset the board, second it should be invoked to run as an actual debugger.

- **Q: Can I program FPGA's in ARC EM Starter Kit or in ARC SDP?**

  OpenOCD has some support for programming of FPGA's over JTAG, however it is not officially supported for ARC development systems.

- **Q: When debugging ARC EM core in AXS101 with Ashling Opella-XD and GDBserver I get an error messages and GDB shows that all memory and registers are zeroes**

  A: Decrease a JTAG frequency to no more than 5MHz using an Ashling GDBserver option `--jtag-frequency`. This particular problem can be noted if GDBserver prints:

  ```
  Error: Core is running (unexpected), attempting to halt...
  Error: Core is running (unexpected), attempting to halt...
  Error: Unable to halt core
  ```

  While GDB shows that whole memory is just zeroes and all register values are also zeroes.

GNU man pages

Man pages for ARC GCC and Binutils can be found at following links:

- GCC man pages
- GNU Assember man pages
- GNU Binutils man pages
- GNU Linker man pages.
- genindex
- search

Index