TargetDB XML Specification - External

Code Composer Studio and dev.ti.com

Exported on 01/19/2024

Table of Contents

1 TargetDB Properties

TargetDB holds all the global information regarding the user's hardware configuration, including:

- ISA level information (e.g. CPU register properties)
- Board level information (e.g. memory maps, memory properties)
- Peripheral names, type, location
- Scan chain ordering
- Emulation and simulation setup information
- Multiple configuration layouts

1.1 TargetDB is an extensible and distributed database

The TargetDB database is spread across multiple XML files. Different XML files are allowed to define different properties of the same database node. Database nodes are identified by their unique path through the XML tree, beginning at the <configurations> root node of the tree down to the leaf nodes.

A node is identified by a combination of its element 'tag' and the value of its 'id' attribute. The fully-qualified path of a node, is the set of tags and ids leading from the root to the node must be unique. Any nodes with the same fully-qualified path of tags and ids will be merged into one node by the TargetDB parser.

The TargetDB parser is designed to merge all the information for a particular node found in multiple XML files into a single data node, this allows for a static XML file, with default values, to be released and a parent XML can then replace its default value if required.

For example, parsing these two files:

file c55_regs.xml ----------

```
<cpu id=”laijin” XML_Version=”1.0”>
<registers id="CPU_Registers">
  <register id=”regs”\>
</registers>
</cpu>
```

file c55_buses.xml ----------

```
<cpu id=”laijin” XML_Version=”1.1”>
<memory id=”internal”\>
</cpu>
```

The resulting XML tree will contain a single <cpu id=”laijin”> node which contains both <registers> and <memory> sub-nodes.

merged result ----------

```
<cpu id=”laijin” XML_Version=”1.1”>
<registers id="CPU_Registers">
  <register id=”regs”\>
  </registers>
  <memory id=”internal”\>
</cpu>
```

The rationale for this feature is to allow a distributed TargetDB to be created. For example, the node describing a peripheral may be filled in by several different files, e.g. from a hardware perspective, a simulator perspective, or an application perspective.

If any of the merged nodes attributes conflict, as the <cpu> XML_Version attribute in the above example does, then the last attribute will be used and no error will be reported.

1.2 TargetDB Directory Structure

By convention, a directory structure is being defined to facilitate easy location and separation of the major sections of the TargetDB structure. The root TargetDB file can exist in any location, the client is responsible for knowing its location and passing this location to the TargetDB parser.

All <include> and <instance> tags should use relative pathnames, this allows the directory structure to be installed in different locations.

The seven sub-directories (drivers, boards, cpus, etc.) shall contain only valid files whose starting tag is the same as the directory name. For example, in the cpus directory all .xml files shall have the <cpu> tag as it’s first element. This applies to files with the .xml extension. This allows setup tools to know what files are valid for each tag level. A setup program can look in the cpus directory and know all .xml files are valid selections for the user. Any other .xml files shall be placed in sub-directories.

TargetDB schema files are called setup_parser.xsd. TargetDB folders are located in ${CCSInstallRoot}\ccs\ccs_base\common\targetdb

The directories structure is:

TargetDB directory structure

```
  ./TargetDB/
Schema files
…user root configuration files

   /connections/
```

```
…all common connection files

   /drivers/
…all common driver files

   /configurations/
…all common platform files

   /boards/
…all common board files

   /devices/
…all common device files

   /cores/
…all common core files

   /cpus/
…all common CPU files

   /routers/
…all common CPU files
```

Explanation of TargetDB structure

```
./TargetDB/

   /connections/
   Files in this directory contain files that describe “connections” that
are used to communicate between debugger and targets. In most cases these
are emulator’s
  that are use to connect to HW targets. One exception is that TI Simulator
is also a connection. Connection files describe properties/options that are
specific
  to a particular connection, thus allowing to configure settings that are
emulator specific.

   /drivers/
   These files define individual drivers that may be used to access specific
processor types on hw boards or simulator drivers. These files also contain
driver
  specific properties that allow to configure how drivers behave.

      /TI_reg_ids/
      Files in this directory contain ISA and/or sub-family files that are
required to convert string id’s to numeric index values that can be used in
register
     access requests passed to the driver. i.e. drivers do not use string
```

```
names for registers, they use numeric values.

   /configurations/
   Configuration files contain nearly complete target configurations that
may be used to ship pre-configured target configurations. These target
configurations
  contain connection information, thus in most cases they are not the ideal
way of distributing target configurations as configuration files will have
to be
  duplicated for every different connection(i.e. emulator). One exception
are simulators, since there is only one type of simulator it makes no
difference if
  device files or configurations are used.

   /boards/
   Board files contain configuration information that is board specific, but
it does not contain connection information. Board files should be used when
complete
  board configurations need to be defined. E.g. EVMs or DSKs as they may
contain multiple devices (i.e. physical chips) on a board.

   /devices/
   Device files contain configuration for a specific device. Device files
should be the primary way of describing configuration information that may
be used by
  debuggers. Device files do not contain any connection information, thus
they may be re-used by different emulators. Device files at minimum should
contain
  complete JTAG layout for a device. They can also optionally have
references to register files which define pseudo and memory mapped
registers. The main
  advantage of specifying pseudo and memory mapped registers in device files
is that register information displayed can be device specific, including all

  specialized peripherals that are on the device.

   /cores/
   Cores are intended to optionally provide a way of providing more
information than CPU, e.g. when an IP module is included in device design
that contains
  more than just CPU. These are not used very frequently.

   /cpus/
   Cpu files contain cpu specific information, they optionally contain all
register information for a particular CPU, including register's that are not
user
   visible e.g. AET/RTDX registers.
```

```
 /routers/
   Router files contain specification and properties for routers. E.g.
ICEPICK, CS_DAP
```

1.3 Working with TargetDB and XML files

It is suggested that user obtain a decent XML editing tool - there are many available. A plain text editor may be used as well for simple edits, but some features that dedicated tools provide are:

- File verification against schema file: this allows to verify that user created .xml file conforms to XML schema i.e. grammar.
- Navigation of schema: this helps to understand the structure to TargetDB and which attributes are required.
- Editor support for XML: syntax highlighting of XML specific files and formatting of XML files.

It is generally a good idea when working with XML files to check that they meet schema specification, by validating xml file being edited against schema file.

1.4 Distributing XML across multiple files

The TargetDB XML parser supports a two new XML tags; <include> and <instance>. The database consists of multiple XML files and these tags allow one XML file to include another. The including is seamless; the included content becomes part of the overall data tree, just the same as if it were defined in place. The <include> tag is a convenience, similar in purpose to the #include feature of the C pre-processor. It is useful for sharing code across XML files, and for breaking up large XML files.

The <instance> tag is similar to the <include> tag with that in addition it copies all attributes in the tag into the first node of the included file. This features allows inclusion of a common file but give it a new ‘id’ tag so it is unique with regards to other nodes.

For example,

File: ./devices/omap1510.xml

```
<device id=”omap1510”>
<processors>
<instance href=”../cpus/c5510_cpu.xml” id=”c55_1” desc=”c5510 device”/>
<instance href=”../cpus/c5510_cpu.xml” id=”c55_2” />
<include href=”./jtag_data.xml” />
</processors>
</device>
```

File: ../cpus/c5510_cpu.xml

```
<cpu id=”TMS320C5510”>
... this node defines the C5510 CPU ...
</cpu>
```

The output of the XML parser will be equivalent to that produced for the following input file:

```
<device id=”omap1510”>
<cpu id=”c55_1” desc=”c5510 device”>
... this node defines the C5510 CPU ...
</cpu>
<cpu id=”c55_2”>
... this node defines the C5510 CPU ...
</cpu>
<jtag ir_length=”8” dr_length=”1” />
</device>
```

1.5 TargetDB attributes

TargetDB is static: The data in the TargetDB is static: it describes unchanging, static properties. Example of TargetDB data: a list of all CPUs on a board. Example of data which does not belong in TargetDB: CCS project information, i.e. which .out files use this configuration.
Since TargetDB expresses only the static information the Parser does not provide write access to the database. All changes need to be made in a new root level file that overrides the static data

TargetDB does not depend on XML content: TargetDB’s Parser does not know the structure of the XML files. For instance, the TargetDB Parser does not know that register information is stored in a <register> element

1.6 General content VS implementation content

One of the concepts of the directory layout and ability to include boards and CPUs into different configurations is that a standard definition of a CPU can be used in several places but defined only once. This works well until you try to define things like off-chip memory within a CPU definition. The problem is that now each time the CPU definition is included in a configuration the off-chip memory definition is also included. This is usually not what is desired.

What is needed is a method of allowing common definitions, like on-chip memory to be defined in the CPU definition and a way to add off-chip memory once the CPU is inserted into a configuration. This can be accomplished by knowing that the XML parser merges all common tags/strings together

Lets use a simplified example to illustrate this concept. In this example we have one CPU (TMS320C6415) with on-chip memory that is included in a configuration that defines an off-chip memory.

File: ./cpu/TMS320C6415.xml

```
<cpu id="c6415" XML_version="6.7" isa="TMS320C6415">
 <memory id="on-chip1">
   <start_address value="0x1000"/>
   <length value="0x4000"/>
 </memory>
</cpu>
```

File: ./platform/myboard.xml

```
<platform id="myboard" XML_version="1.0">
<board id="myboard" XML_version="1.3">
<instance href=”../cpu/TMS320C6415.xml” id=”Kelvin/>
<cpu id="Kelvin">
  <memory id="off-chip1">
    <start_address value="0x5000"/>
    <length value="0x4000"/>
  </memory>
</cpu>
</board>
</platform>
```

The resulting XML document will be:

```
<platform id="myboard" XML_version="1.0">
 <board id="myboard" XML_version="1.3">
    <cpu id="Kelvin" XML_version="6.7" isa="TMS320C6415">
       <memory id="on-chip1">
           <start_address value="0x1000"/>
           <length value="0x4000"/>
       </memory>
       <memory id="off-chip1">
           <start_address value="0x5000"/>
           <length value="0x4000"/>
       </memory>
   </cpu>
</board>
</platform>
```

2 CCS v4+ : Use of TargetDB

CCSv4 and up uses targetdb for two main purposes. Definition of building blocks required to construct target configuration files, which are used to specify targets for debugging and optionally to specify register sets that debugger will use for display in register window.

2.1 TargetDB organization in CCSv4+

TargetDB organization in CCSv4+

```
 ./TargetDB/

   /connections/
    Files in this directory contain files that describe “connections” that
are used to communicate between debugger and targets. In most cases
    these are emulator’s that are use to connect to HW targets. One
exception is that TI Simulator is also a connection. Connection files
    describe properties/options that are specific to a particular
connection, thus allowing to configure settings that are emulator specific.

   /drivers/
   These files define individual drivers that may be used to access specific
processor types on hw boards or simulator drivers. These files also
   contain driver specific properties that allow to configure how drivers
behave.

   /configurations/
   Configuration files contain nearly complete target configurations that
may be used to ship pre-configured target configurations. These
   target configurations contain connection information, thus in most cases
they are not the ideal way of distributing target configurations
   as configuration files will have to be duplicated for every different
connection(i.e. emulator). One exception are simulators, since there
   is only one type of simulator it makes no difference if device files or
configurations are used.

   /boards/
   Board files contain configuration information that is board specific, but
it does not contain connection information. Board files should
   be used when complete board configurations need to be defined. E.g. EVMs
or DSKs as they may contain multiple devices (i.e. physical chips) on a
board.

   /devices/
```

```
   Device files contain configuration for a specific device. Device files
should be the primary way of describing configuration information
   that may be used by debuggers. Device files do not contain any connection
information, thus they may be re-used by different emulators.
   Device files at minimum should contain complete JTAG layout for a device.
They can also optionally have references to register files which
   define pseudo and memory mapped registers. The main advantage of
specifying pseudo and memory mapped registers in device files is that
   register information displayed can be device specific, including all
specialized peripherals that are on the device.

   /cores/
   Cores are intended to optionally provide a way of providing more
information than CPU, e.g. when an IP module is included in device design
   that contains more than just CPU. These are not used very frequently.

   /cpus/
   Cpu files contain cpu specific information, they optionally contain all
register information for a particular CPU, including register's
   that are not user visible e.g. AET/RTDX registers.

   /routers/
   Router files contain specification and properties for routers. E.g.
ICEPICK, CS_DAP
```

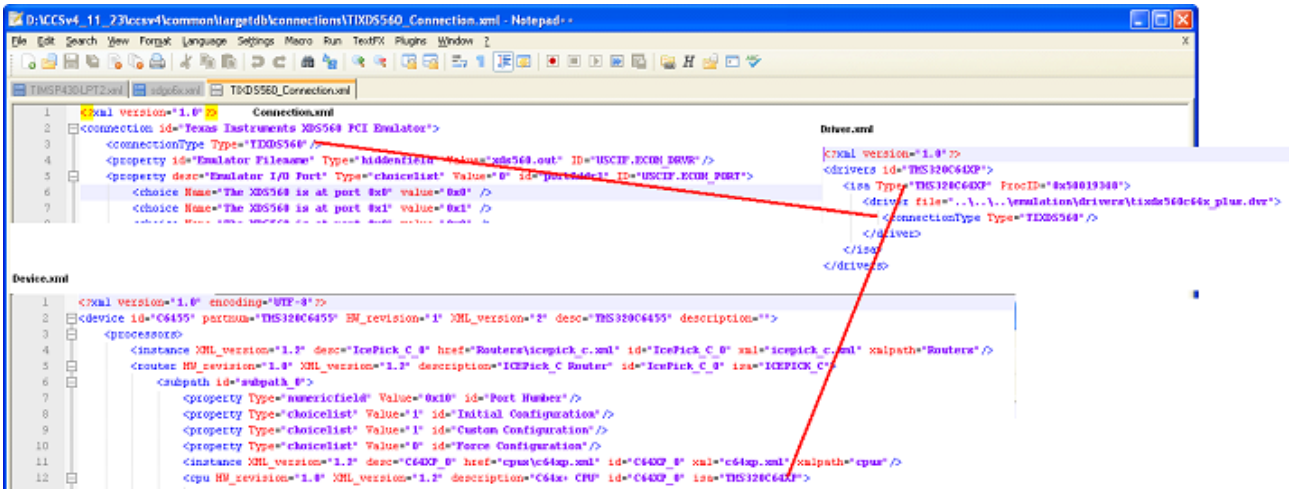
2.2 CCS Setup Editor use of TargetDB

CCS Setup Editor makes all its options available in user interface by reading all TargetDB files and constructing internal structures that are used to cross reference and eliminate options that are not valid.

Basic tab of target configuration editor provides selection of connection file as the first option that user needs to select. Options listed in connections combo box are obtained by including all valid connection files. <connection id="String" String is used to display a connection to user in setup editor.

List of boards/devices presented to the user is computed based on user selected connection file. The algorithm to determine list of boards/devices works as follows

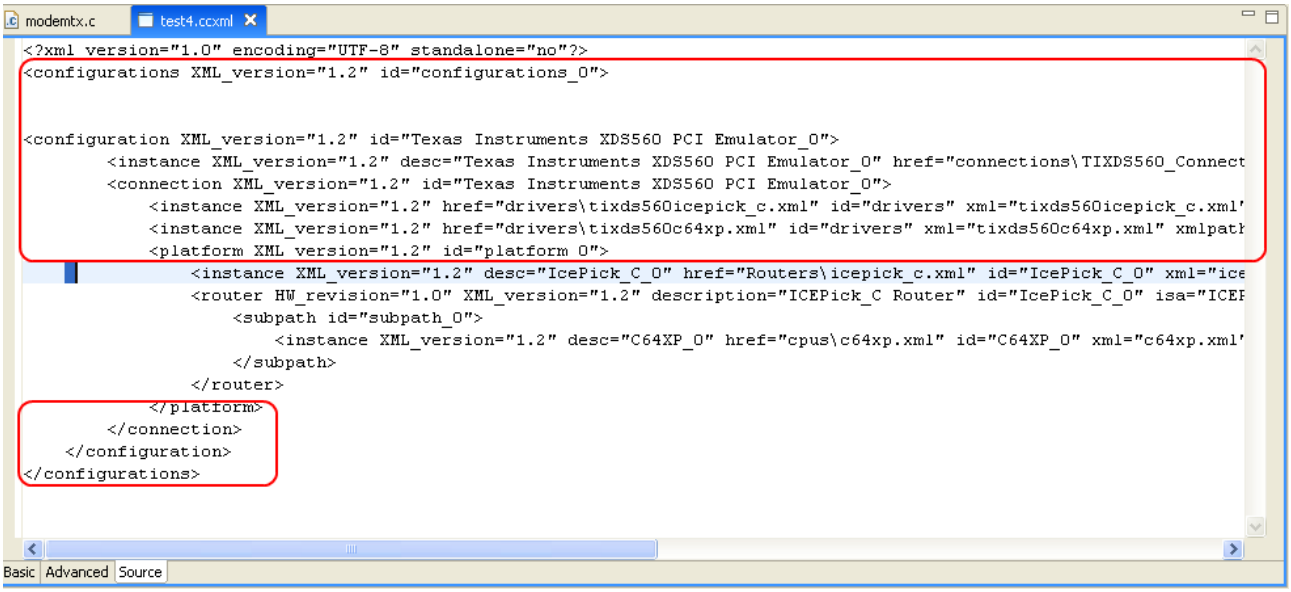
- Use <connectionType Type attribute in connection.xml file to identify all driver.xml files that support that particular connection(i.e. use <connectionType Type attribute in driver.xml file)
- From selected driver.xml files build a list of isa Types that can be used to filter out board/device files. (use <isa Type attribute from driver.xml file)
- Using a list of isa Type identified in previous step to correlate with <cpu nodes in boards/device files. i.e. specifically use isa attribute of <cpu node to match it with isa Types.
- In Setup editor display a list of boards/device files where all cpus and routers have a valid driver for that connection type. (i.e. match <cpu nodes isa attribute with drivers isa Type



2.3 Device Files

Device files are XML files that define JTAG layout for a particular device. They can also optionally contain register definitions. This is the recommended way of adding new device support to CCS, unless it is a complete board, in which case a board file should be created. The easiest way to create a device file in CCSv4 is to use setup editor by creating New Target configuration. Select Target->New Target Configuration and enter a name for your configuration(it does not matter what this file is called as it will be used only to get xml content for a device file). Switch to Advanced tab and click on new to add an emulator and then click Add to add ICEPICK-C, subpaths and cpus. The specific items to be added will be dependent on the devices JTAG layout. Once you are satisfied with JTAG layout specification and all properties have been configured save the target configuration file. We will now need to convert this complete target configuration file into a device file.

Open .ccxml file that was created in previous step using a text editor and save it as deviceName.xml. (You can find out the location of .ccxml file by opening “Target Configuration” view, select and right click on your .ccxml file and select properties). We will need to remove some emulator specific information to make this device file emulator agnostic. Remove <platform node and everything above it, except first line. Also, remove </platform node and everything below it.



Then add <device node on second line and </device> on the last line.

The <device node needs to have following attributes that are adjusted to match your device

<device id="DM6437" partnum="TMS320DM6437" HW_revision="1" XML_version="2" desc="TMS320DM6437" description="DaVinci">

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<device id="DM6437" partnum="TMS320DM6437" HW_revision="1" XML_version="2" desc="TMS320DM6437" description="DaVinci">
    <instance XML_version="1.2" desc="IcePick_C_0" href="Routers\icepick_c.xml" id="IcePick_C_0" xml="icepick_c.xml" xmlpath="Routers"/>
    <router HW_revision="1.0" XML_version="1.2" description="ICEPick_C Router" id="IcePick_C_0" isa="ICEPICK_C">
        <subpath id="subpath_0">
            <instance XML_version="1.2" desc="C64XP_0" href="cpus\c64xp.xml" id="C64XP_0" xml="c64xp.xml" xmlpath="cpus"/>
        </subpath>
    </router>
</device>
```

As a last step you will need to drop your deviceName.xml file into <CCS_INSTALL_ROOT>/common/targetdb/devices folder and restart CCS. Your device should now appear in Device selection control in setup editor.

2.4 Peripheral registers in Device files

CCS v4 can optionally read in definitions for registers that should be displayed for a particular device from XML files. If register definitions are not specified in XML files then a binary component provides register information that contains a relatively generic set of registers that is applicable to a processor type. The main advantage of using XML files to specify register definitions is that they can display registers that are device specific and include registers for all peripherals on the device. The information is also more detailed as XML files usually define bitfield information in more detail and provide short description on what register and individual bitfields function.

Below is a snippet of a device file with some attributes/additional data removed to help illustrate where to add peripheral register information. Open device file created in previous step in a text editor.

1. Find the cpu definition in device file where peripheral register definitions need to be added. In the sample below we are adding peripheral information to 64x+ cpu of a DM6435 device.

a. In a multi-processor device you will need to do this for every processor adjusting included peripherals and base address as necessary to match device specification.

2. You will need to adjust XML to be able to add child nodes to cpu node. By default cpu node will look like

<cpu HW_revision="1.0" XML_version="1.2" description="C64X+ CPU" id="C64XP_0" isa="TMS320C64XP"/>

It needs to be split up into 2 separate nodes to allow addition of peripheral registers. If you modified some CPU properties during creation of a device file then there may already be two cpu nodes created

<cpu HW_revision="1.0" XML_version="1.2" description="C64X+ CPU" id="C64XP_0" isa="TMS320C64XP">

----content can now go here----

</cpu>

3. We can now insert <instance statements that will include peripheral definition files into device files. See lines in red for example of all necessary attributes. Key attributes are
href – this specifies relative path to definition of peripheral register xml file.
baseaddr – this attribute specifies the base address where peripheral is mapped into devices memory. Peripheral files contain only the offset from the baseaddr, thus allowing peripheral specification files to be re-used by different devices.

```
  <device id="DM6435" partnum="TMS320DM6435" HW_revision="1" …>
<instance XML_version="1.2" desc="IcePick_C_0"
href="Routers\icepick_c.xml"… />
<router HW_revision="1.0" XML_version="1.2" description="ICEPick_C Router"…>
   <subpath id="Subpath_0">
      <instance XML_version="1.2" desc="C64XP_0" href="cpus\c64xp.xml" .../>
      <cpu HW_revision="1.0" XML_version="1.2" description="C64X+ CPU"
id="C64XP_0" isa="TMS320C64XP">
         <instance href="..\Modules\dm64lc_cslr_intc.xml" id="DSPINTC"
xml="dm64lc_cslr_intc.xml" xmlpath="..\Modules\" HW_version="DaVinci"
description="DSP Interrupt Controller" requestor="TMS320C64XX"
baseaddr="0x1800000" endaddr="0x180ffff" size="0x10000" accessnumbytes="4"
permissions="p" />
         <instance href="..\Modules\dm64lc_cslr_idma_001.xml" id="IDMA"
xml="dm64lc_cslr_idma_001.xml" xmlpath="..\Modules\" HW_version="DaVinci"
description="DSP Internal DMA Controller" requestor="TMS320C64XX"
baseaddr="0x1820000" endaddr="0x18201ff" size="0x200" accessnumbytes="4"
permissions="p" />
         …
         …
      </cpu>
```

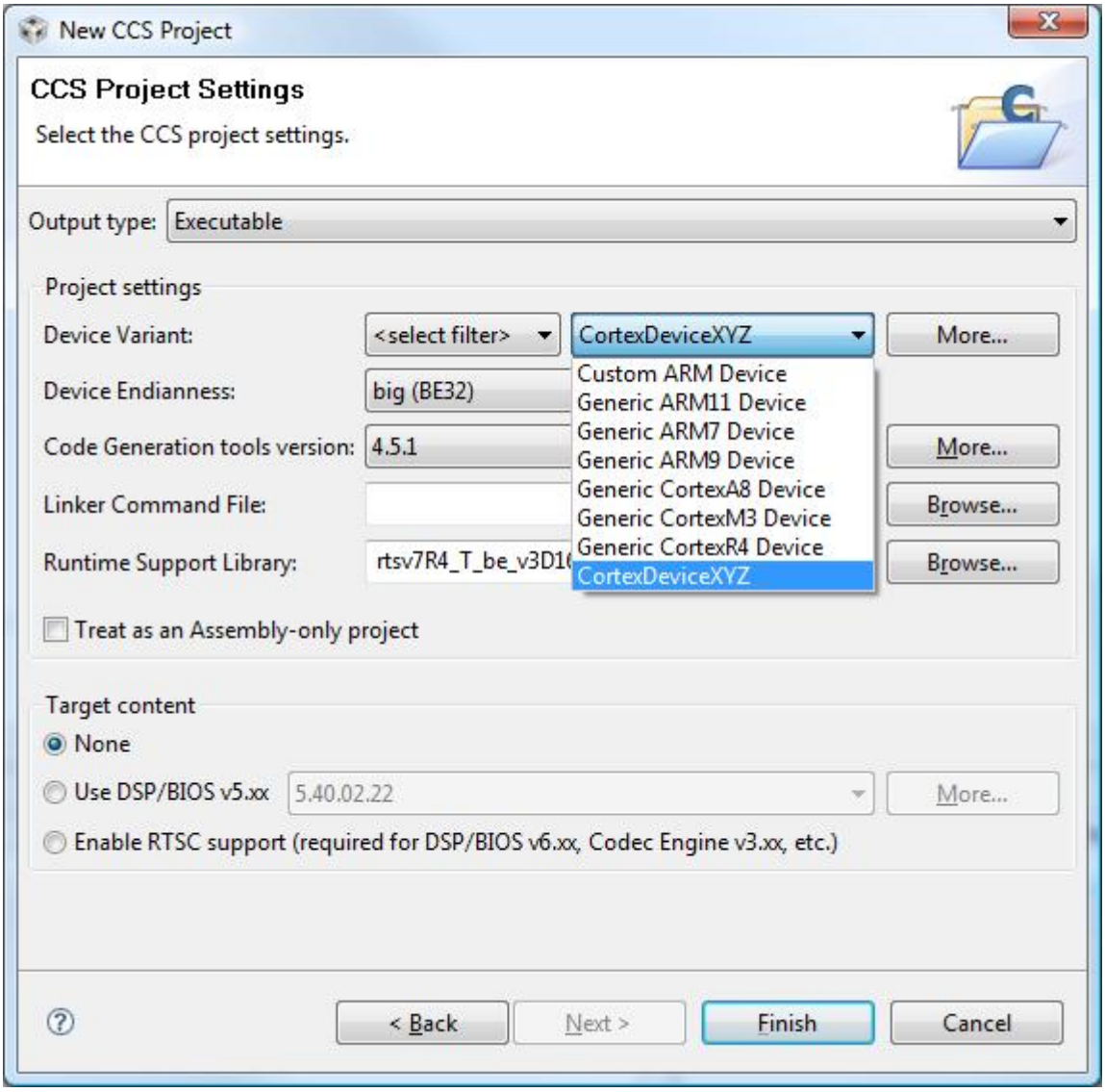
NOTE: Above scheme is not applicable to CCSv3.3. CCS v3.3 device.xml files do not contain JTAG layout information and only registers are defined. The formatting is different, thus 3.3 device files are not usable in CCSv4.

2.5 Pseudo Registers

Files under cpus or routers specify register names that need to be shown or made available to users. Files under TI_reg_ids folder contain string to register index mapping that the driver will understand (string to a number mapping as GTI does not use strings for register accesses). Both files(i.e. files under cpus and/or routers and files under TI_reg_ids folder) are required, thus all registers defined in either a cpu or router need to be also defined in appropriate ti_reg_ids file. If they are not, then most likely you will get a message from CCS saying that register xyz could not be resolved. The naming convention of ti_reg_ids files needs to be followed exactly. It needs to be “decodedProcIDString_regids.xml”, the debugger also supports wild cards. E.g. there is a TMS470R2X_regids.xml file that is used for ARM9, however if this file did not exist then TMS470RXX_regids.xml would be used as it is more generic(i.e. no subfamily is specified).

2.6 Project options in Device Files

Device XML files may contain additional information that will be used during project creation. Device specific compiler or linker options may be specified. Additionally, device file may specify standard library to be used or linker command file. During new project creation if user selects a specific device, then compiler, linker, library, linker cmd options specified in device file will be used to set project settings. See second picture for an example of how to specify various options through properties specifed in device file.



Supported project options are listed below. A device file may specify all, a subset or none of the options listed below. Example of syntax is provided in sample device file shown below(highlighted in red).

CompilerBuildOptions and **LinkerBuildOptions** Value field should be set to a list of space separated tool flags.

LinkerCmd and **RTSlib** accept a string specifying a file name. The Linker-command file with this name will be looked up, during project creation, in the <install>/ccsv4/<isa-family>/include/ directory, where <isa-family> is the ISA family in lowercase, eg. "msp430". The RTS-library file-name will be added to the linker's library-list as is, and will be looked up, during project build, along the linker search-path. Valid options of <isa-family> are c2000, msp430, c5400,c5500, c6000, arm.

MinCodegenVersion accepts a string specifying the minimum version of the codegen tool that can support this device. The New Project wizard will prevent the user from creating a project for such device and a lower than the minimum version of a codegen tool.

IsElfDefault accepts a boolean specifying if the binary output-format should be "elf" by default. If this property is omitted, the output-format is by default "coff".

```
<device id="CortexDeviceXYZ" partnum="CortexDeviceXYZ" HW_revision="1.0"
XML_version="1.0" description="CortexDeviceXYZ">
<instance XML_version="1.2" desc="IcePick_C_0" href="Routers\icepick_c.xml"
id="IcePick_C_0" xml="icepick_c.xml" xmlpath="Routers"/>
<router HW_revision="1.0" XML_version="1.2" description="ICEPick_C Router"
id="IcePick_C_0" isa="ICEPICK_C">
<subpath id="subpath_0">
<instance XML_version="1.2" desc="Cortex_R4_0" href="cpus\cortex_r4.xml"
id="Cortex_R4_0" xml="cortex_r4.xml" xmlpath="cpus"/>
<cpu HW_revision="1.0" XML_version="1.2" description="Cortex_R4 CPU"
id="Cortex_R4_0" isa="Cortex_R4">
 <property Type="stringfield" Value="--silicon_version=7R4 --
float_support=VFPv3D16 --abi=eabi" id="CompilerBuildOptions" />
<property Type="stringfield" Value="--be32" id="LinkerBuildOptions" />
<property Type="stringfield" Value="link.cmd" id="LinkerCmd" />
<property Type="stringfield" Value="rtsv7R4_T_be_v3D16_eabi.lib"
id="RTSlib" />
<property Type="stringfield" Value="4.5.1" id="MinCodegenVersion" />
<property Type="stringfield" Value="true" id="IsElfDefault" />
<property Type="filepathfield" Value="..\..\..
\emulation\gel\CortexR4_util.gel" id="GEL File" />
<property Type="numericfield" Value="-2147479552" id="Address" />
</cpu>
</subpath>
</router>
</device>
```