Config bld basics

Here's a basic overview of the config.bld files that appear in the DVSDK and with Codec Engine-based projects. A quick search shows config.bld in the DVSDK demos, Codec Engine examples, codec combos, DMAI examples, DVTB, and DSPLink. Some of these config.bld are generated by the project authors while others are generated automatically by configuro. Chances are that if you're writing an application using Codec Engine or generating a RTSC package using the RTSC Wizards, you'll need to supply your own config.bld, so we've tried to condense a few dozens resources into a simple config.bld FAQ/tutorial.

Contents

What is config.bld and why do I need it?

Each "project" will (usually) have its own config.bld. Thus each "project author" should provide their own config.bld else future users will either have to create their own or rely on another config.bld in their system (which may lead to unexpected results--you'll see why below). A "project" may consist of multiple applications - perhaps some running on the ARM and some on the DSP. Coming from a CCS perspective, you may want to think of what kinds of information you normally have in your "CCS Build Options"--such as the intended build target, compiler options, etc. For example, the IUNIVERSAL Examples contain two codecs (fir and randgen) that can each run on either the ARM or DSP, multiple servers (for DM6446, OMAP3530, and OMAPL137), and an application to run each codec--but since we expect a single user to rebuild the codecs, servers, and apps, we provide only a single config.bld for the IUNIVERSAL Examples project.

A config.bld is a file needed by the XDCTools that simply associates targets and platforms. A target can be thought of as a compiler and compiler options. For example, you might have a 64x+ target - which would use the TI C6000 Code Generation Tools with appropriate compiler flags to indicate that it should build for a 64x+ DSP. The XDCTools predefines a set of targets--to see the list of TI targets, see xdctools_x_xx/packages/ti/targets; to see the GNU targets, see xdctools_x_xx/packages/gnu/targets, etc. This tends to make a lot more sense if you look at a target file, such as xdctools_x_xx/packages/ti/targets/C64P.xdc. This file shows all the default options for the 64x+ target. For each target, we need to then associate a platform--which can be thought of as the actual hardware. For example, platforms include ti.platforms.evmDM6446 and ti.platforms.evm3530 (for DM6446 and OMAP3530 respectively). These platforms are found in xdctools_x_xx/packages/ti/platforms. Note that the directory structure corresponds to the names of the targets and platforms where "/" are exchanged for ".".

Without a config.bld, we would need to specify which compiler to use, the path to the compiler (on the host machine), which compiler options to use, etc. in every application. The config.bld allows each project to simply do it once.

Which config.bld is used when I build? How can I specify which one to use?

The XDCTools searches for the config.bld file in your current directory, followed by the paths on your XDCPATH. This frequently leads to the incorrect config.bld being used (i.e. resulting in building for an unintended target which may be entirely unsupported, etc.) thus you may find it beneficial to explicitly name the config.bld to be used.

If using the *xdc* build command directly, add the XDCBUILDCFG option as follows:

```
$(XDC) XDCBUILDCFG=$(PACKAGE_DIR)/config.bld
```

If using configuro, use the -b option and pass the path to the desired config.bld. Usually define the top-level location of your project and keep the config.bld there (indicating its use throughout the entire project in the directories beneath it). In the following example, we predefined $(PACKAGE_DIR) as our project's top-level directory.

```
XDCPATH="$(XDCPATH)" $(CONFIGURO) -o $(XDC_CFG) -t $(XDC_ARM_TARGET) -p $(XDC_PLATFORM) -b $(PACKAGE_DIR)/config.bld $(XDC_CFGFILE)
```

In case the above command looks cryptic, note the above variables are set in the Makefile as follows: CONFIGURO = $(XDC_INSTALL_DIR)/xs xdc.tools.configuro - command to start configuro

- `XDC_CFG = cfg/$(MY_APPLICATION_NAME)` - Path to output configuration files
- `XDC_ARM_TARGET = gnu.targets.arm.GCArmv5T` - XDC ARM target definition (see definition of target above)
- `XDC_PLATFORM = ti.platforms.evmDM6446` - XDC platform definition
- `XDC_CFGFILE = $(MY_APPLICATION_NAME).cfg` - Input configuration file (For ARM-side codecs, this file would include your framework component (e.g. DSKT2) configuration and Engine.create(). For DSP-side codecs, this file would include your `Engine.createFromServer()`.)

Thus we're simply invoking configuro to generate the necessary XDC files in a certain location for a given target, platform, and configuration.

What is user.bld and what does this have to do with config.bld?

The Codec Engine examples run on almost every TI platform (DM6446, OMAP3530, DM355, DM6467, and the list goes on). These platforms can use a variety of compilers - MV 4.0, MV 5.0, Code Sourcery, etc. Thus to support the permutations of platforms and compilers, the user.bld file was created. The Codec Engine example's config.bld then explicitly specifies that it should use the content from user.bld. We should be very clear that the user.bld is not required by generic config.bld files. This Codec Engine config.bld/user.bld idea works well if you need to run your code on many platforms, but it's probably overkill if you only want it to run on one or two platforms. Supposing that that's the case, we don't recommend using the Codec Engine config.bld/user.bld - but instead using the examples below. If you really insist on seeing some "live" examples, see the IUNIVERSAL Examples (that use Codec Engine) available in this article.

How do I just specify a DSP target (e.g. what's a very simple config.bld look like)?

Below we simply define a target and associate the location of the compiler (the path to the C6000 toolchain) and a platform (in this case, ti.platforms.evm3530) with that target.

config.bld for 64P only <syntaxhighlight lang='javascript'> /* Get a handle to the C64P Target */ var C64P = xdc.useModule('ti.targets.C64P');

/* Configure the C64P target with the location of your C6000 tools */ C64P.rootDir="/opt/cg6x_6_0_16";

/* Associate a platform with this target */ C64P.platform = "ti.platforms.evm3530";

/* Add the C64P target to the Build.targets array */ Build.targets.$add(C64P); </syntaxhighlight>

Notice that the `Build.targets` array specifies which targets to build for. The example above uses the `$add()` method to add an element to the array (this approach makes the snippet above a little more standalone and cut-n-paste-able into other config.bld's). Sometimes the array is explicitly populated like this:

<syntaxhighlight lang='javascript'> Build.targets = [

```
  C64P,
```

]; </syntaxhighlight>

Which to use is personal preference, but you'll encounter both.

The `Build.targets` array is accessible via each package build script (AKA package.bld) in your project. For example, your codec may build for ARM and DSP whereas your server would only build for DSP, thus the package.bld can add a statement as follows:

package.bld snippet - indicating that a library should be produced for each target listed in Build.targets <syntaxhighlight lang='javascript'> for (var i = 0; i < Build.targets.length; i++) {

```
  var targ = Build.targets[i];
  var lib;
```

```
  /*
   * Pkg.addLibrary() adds a library named lib/fir_ti.<target_specific_suffix>
   * to this package.
   */
  lib = Pkg.addLibrary("lib/fir_ti", targ, {
      suffix: ".l" + targ.suffix
  });
```

```
  /* Add the sources defined in the SRCS array to the library */
  lib.addObjects(SRCS);
```

} </syntaxhighlight>

package.bld snippet - indicating that a DSP server should only be produced if the OS is undefined (implying DSP/BIOS) <syntaxhighlight lang='javascript'> for (var i = 0; i < Build.targets.length; i++) {

```
  var targ = Build.targets[i];
  var exe;
```

```
  if (targ.os == undefined) {
      /* Assume DSP/BIOS */
```

```
      /*
       * Pkg.addExecutable() builds an executable named serverName.<target_specific_suffix>
       * for the target "targ" and platform "targ.platform" with the given config options
       */
      exe = Pkg.addExecutable(serverName, targ, targ.platform, {
          tcopts: "-Dxdc.cfg.check.fatal=false",  // if necessary, disable checks  :(
          cfgScript: "server.tcf",                // BIOS 5-based exe's require a .tcf script
          profile: arguments[0],                  // Build for the profile passed into this script
          lopts: "-l link.cmd",                   // Use the link.cmd linker command file
      });
```

```
      /* Add the "main.c" source file to the executable */
      exe.addObjects(["main.c",]);
  }
```

} </syntaxhighlight>

For more information on the package build script, see the RTSC Packaging Primer (http://rtsc.eclipse.org/docs-tip/RTSC_Packaging_Primer).

What are the different types of ARM targets?

In the beginning, there were only ARM9 platforms and MontaVista targets, so we created config.bld like this (appropriate for DM6446, DM355, etc.). Note that this is analogous to the DSP-only config.bld above--that is, specify the target, path to the compiler and associate a platform to the target.

config.bld for DM6446 ARM9 only - using the MVArm9 target <syntaxhighlight lang='javascript'> var MVArm9 = xdc.useModule('gnu.targets.MVArm9'); MVArm9.rootDir = "/opt/mvl5_0/montavista/pro/devkit/arm/v5t_le"; MVArm9.platform = "ti.platforms.evmDM6446";

/* Add the MVArm9 target to the Build.targets array */ Build.targets.$add(MVArm9); </syntaxhighlight>

Later, the toolchains diversified and there were non-ARM9 ARM platforms, so the "Generic ARM" target was created. This target would then work for both ARM9 (using either MV or CodeSourcery toolchains) as well as Cortex-A8. Since not all ARM toolchains have the same prefix, additional information is needed in the config.bld as follows. In this example, we combine both the ARM and DSP configuration to provide a complete config.bld for OMAP3530.

config.bld for OMAP3530 - builds both for the DSP as indicated by C64P target and for the ARM (via Generic ARM target) <syntaxhighlight lang='javascript'> /* DSP Configuration */ var C64P = xdc.useModule('ti.targets.C64P'); C64P.rootDir="/opt/ti/c6x/6.0.16/Linux" C64P.platform = "ti.platforms.evm3530";

Build.targets.$add(C64P);

/* Generic ARM Configuration */ var GCArmv5T= xdc.useModule('gnu.targets.arm.GCArmv5T'); /* Name of XDCTools Generic ARM Target */ GCArmv5T.rootDir = "/opt/toolchain/arm-2007q3"; /* Location of toolchain (path) */ GCArmv5T.LONGNAME = 'bin/arm-none-linux-gnueabi-gcc'; /* Each toolchain may have different prefix, this is the actual executable */ GCArmv5T.platform = "ti.platforms.evm3530"; /* Associate the target with a platform */

/* remove reference to C++ from opts */ GCArmv5T.lnkOpts.suffix = GCArmv5T.lnkOpts.suffix.replace("-lstdc++", ""); /* For example add a linker option */

Build.targets.$add(GCArmv5T); </syntaxhighlight>

Note that in all of the above examples, the "rootDir" variable must be set appropriately for the host computer. If you, your colleagues, or your customers are uncomfortable editing this file, you can automate it as follows.

Can I create a config.bld that doesn't need to be edited by the end user (i.e. pull in environment variables)?

Yes - you can tie your config.bld to a specific Makefile (and potentially Rules.make). The idea is that your Makefile should export (in Linux) or set (in Windows) specific variables that are then used in the config.bld. This is best illustrated by example:

Makefile snippet (no goals) - to illustrate how to set variables in Makefile for consumption by config.bld <syntaxhighlight lang='make'>

1. Includes your Rules.make from the DVSDK directory, usually dvsdk_x_xx_xx/Rules.make
2. Now the variables defined in the Rules.make are accessible within this Makefile

- include /home/dvsdk_x_xx_xx/Rules.make

1. Location of TI Code Generation Tools if not provided in Rules.make

ifndef CODEGEN_INSTALL_DIR

```
CODEGEN_INSTALL_DIR = /home/cg6x_6_1_7
```

endif

1. Add an if statement to select the appropriate platform if needed

ifeq ($(PLATFORM),dm6446)

```
XDC_PLATFORM = ti.platforms.evmDM6446
XDC_TARGETS = "gnu.targets.arm.GCArmv5T ti.targets.C64P"
CROSS_COMPILE = $(MVTOOL_PREFIX)
```

else ifeq ($(PLATFORM),omap3530)

```
XDC_PLATFORM = ti.platforms.evm3530
XDC_TARGETS = "gnu.targets.arm.GCArmv5T ti.targets.C64P"
CROSS_COMPILE = $(CSTOOL_PREFIX)
```

else ifeq ($(PLATFORM),omapl137)

```
XDC_PLATFORM = ti.platforms.evmOMAPL137
XDC_TARGETS = "gnu.targets.arm.GCArmv5T ti.targets.C674"
CROSS_COMPILE = /library/cs/arm-2007q3/bin/arm-none-linux-gnueabi-
```

else

1. You will have to create servers for your platform and target and set
2. the below variables accordingly, see the iuniversal example design doc.

```
XDC_PLATFORM =
XDC_TARGETS =
CROSS_COMPILE =
```

endif endif endif

1. Parse XDC_TARGETS to find the ARM and DSP targets
2. Note that the XDC_TARGETS is defined as a string that is space separated
3. e.g. "gnu.targets.arm.GCArmv5T ti.targets.C64P" -> XDC_ARM_TARGET="gnu.targets.arm.GCArv5T", etc.

XDC_ARM_TARGET = $(shell echo $(XDC_TARGETS) | cut -d' ' -f1) XDC_DSP_TARGET = $(shell echo $(XDC_TARGETS) | cut -d' ' -f2)

1. Export environment variables needed by config.bld
2. Once exported, the config.bld can use Java to access these variables
3. e.g. var dsptarget = "" + java.lang.System.getenv("XDC_DSP_TARGET");

export CODEGEN_INSTALL_DIR export XDC_PLATFORM export CROSS_COMPILE export XDC_ARM_TARGET export XDC_DSP_TARGET

1. Fill in your Makefile goals here...

</syntaxhighlight>

config.bld that uses environment variables set in the above Makefile <syntaxhighlight lang='javascript'> /* location of your C6000 codegen tools */ var dsptarget = "" + java.lang.System.getenv("XDC_DSP_TARGET"); var codegen = "" + java.lang.System.getenv("CODEGEN_INSTALL_DIR"); var xdcplat = "" + java.lang.System.getenv("XDC_PLATFORM");

var C6X = xdc.useModule(dsptarget); C6X.rootDir = codegen; C6X.platform = xdcplat;

/* User passes in $(CROSS_COMPILE) where $(CROSS_COMPILE)gcc is their compiler

```
Then the TOOLDIR and LONGNAME are derived based on a regex of CROSS_COMPILE
```

- /

var crosscompile = "" + java.lang.System.getenv("CROSS_COMPILE");

var tooldir = ""; var longName = "";

/* Search CROSS_COMPILE for bin/ If only 1 bin/ is found, set the tooldir to

```
the path prior to bin/ and the prefix to "bin/" + remainder of path,
else leave the tooldir as "" and set the LONGNAME to the full CROSS_COMPILE
path
```

- /

var regex = new RegExp("bin/"); var find = crosscompile.split(regex);

if (find[0]!=crosscompile && find.length==2) {

```
tooldir = find[0];
longName = "bin/" + find[1] + "gcc";
```

} else {

```
longName = crosscompile + "gcc";
```

}

/* location of the GCC Arm9 tools */ var GCArmv5T = xdc.useModule('gnu.targets.arm.GCArmv5T'); GCArmv5T.LONGNAME = longName; GCArmv5T.platform = java.lang.System.getenv("XDC_PLATFORM");

- GCArmv5T.rootDir = tooldir;

/* intentionally empty */ Build.targets = []; </syntaxhighlight>

You can even add checks within the config.bld in JavaScript to verify that the rootDir are legal paths. A complete example is provided below:

config.bld with sanity checks on paths <syntaxhighlight lang='javascript'> /* location of your C6000 codegen tools */ var dsptarget = "" + java.lang.System.getenv("XDC_DSP_TARGET"); var codegen = "" + java.lang.System.getenv("CODEGEN_INSTALL_DIR");

/* Verify that XDC_TARGETS was set correctly as an environment variable */ if (dsptarget=="null" || dsptarget=="") {

```
  print("Warning: XDC_DSP_TARGET  not found. Verify that "
      + "XDC_TARGETS is set correctly in the Makefile.");
```

}

/* Verify that CODEGEN_INSTALL_DIR was set as an environment variable */ if (codegen=="null" || codegen=="") {

```
  print("Warning: TI Code Generation Tools not found. Verify that "
      + "CODEGEN_INSTALL_DIR is set correctly in the Makefile.");
```

} else if (!java.io.File(codegen).isDirectory()) {

```
  print("Warning: CODEGEN_INSTALL_DIR is not set to a valid directory.");
```

} /* Verify that XDC_PLATFORM was set as an environment variable */ var xdcplat = "" + java.lang.System.getenv("XDC_PLATFORM"); if (xdcplat=="null" || xdcplat==""){

```
  print("Warning: XDC_PLATFORM not found. Verify that XDC_PLATFORM "
      + "is set correctly in the Makefile.");
```

}

var C6X = xdc.useModule(dsptarget); C6X.rootDir = codegen; C6X.platform = xdcplat;

/* User passes in $(CROSS_COMPILE) where $(CROSS_COMPILE)gcc is their compiler

```
 Then the TOOLDIR and LONGNAME are derived based on a regex of CROSS_COMPILE
```

  - /

var crosscompile = "" + java.lang.System.getenv("CROSS_COMPILE");

/* Verify that CROSS_COMPILE was set as an environment variable and that

```
 $(CROSS_COMPILE)gcc is a valid file */
```

if (crosscompile=="null" || crosscompile=="") {

```
  print("Warning: ARM toolchain not found. CROSS_COMPILE is currently set "
      + "to " + crosscompile + ". Verify that CROSS_COMPILE has been set "
      + "correctly in the Makefile.");
```

} else if (!java.io.File(crosscompile + "gcc").exists()) {

```
  print("Warning: GCC compiler " + crosscompile + "gcc does not exist. "
      + "Verify that CROSS_COMPILE has been set correctly in the Makefile");
```

}

var tooldir = ""; var longName = "";

/* Search CROSS_COMPILE for bin/ If only 1 bin/ is found, set the tooldir to

```
 the path prior to bin/ and the prefix to "bin/" + remainder of path,
 else leave the tooldir as "" and set the LONGNAME to the full CROSS_COMPILE
 path
```

  - /

var regex = new RegExp("bin/"); var find = crosscompile.split(regex);

if (find[0]!=crosscompile && find.length==2) {

```
  tooldir = find[0];
  longName = "bin/" + find[1] + "gcc";
```

} else {

```
  longName = crosscompile + "gcc";
```

}

/* location of the GCC Arm9 tools */ var GCArmv5T = xdc.useModule('gnu.targets.arm.GCArmv5T'); GCArmv5T.LONGNAME = longName; GCArmv5T.platform = java.lang.System.getenv("XDC_PLATFORM"); GCArmv5T.rootDir = tooldir;

Build.targets = [ ];

</syntaxhighlight>

Note that the Build.targets is intentionally left empty here. Instead of indicating which targets to build for in the config.bld, this is done so in the Makefile via XDC option XDCTARGETS:

- <syntaxhighlight lang='make'>

1. Note that we earlier set XDC_TARGETS = "gnu.targets.arm.GCArmv5T ti.targets.C64P"

XDC = $(XDC_INSTALL_DIR)/xdc XDCOPTIONS=$(XDCOPTIONS) XDCTARGETS=$(XDC_TARGETS) </syntaxhighlight>

This granularity would be useful if you intended to build some codecs only for DSP and others for ARM. Again it's useful to examine the IUNIVERSAL Examples config.bld and Makefile to see how they relate to each other.

Note that this example is for a platform with both ARM and DSP cores. It can be modified (say for DM365--ARM only) by removing all config.bld setup for the DSP target (such as any C6X lines and the JavaScript error checking) with appropriate modification in the Makefile as well.

What if I need more information

- RTSC Packaging Primer on config.bld (http://rtsc.eclipse.org/docs-tip/RTSC_Packaging_Primer/Lesson_5)
- RTSC doc on Managing Compiler Toolchains (http://rtsc.eclipse.org/docs-tip/Managing_Compiler_Toolchains)
- RTSC doc on Creating Targets (http://rtsc.eclipse.org/docs-tip/Creating_Targets)

{{

1. switchcategory:MultiCore=

- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **Config bld basics** here.

Keystone=

- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Please post only comments related to the article **Config bld basics** here.

C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article **Config bld basics** here.*

DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article **Config bld basics** here.*

MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article **Config bld basics** here.*

OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Config bld basics** here.*

OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Config bld basics** here.*

MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article **Config bld basics** here.*

*For technical support please post your questions at http://e2e.ti.com. Please post only comments about the article **Config bld basics** here.*

}}

Links

Amplifiers & Linear
Audio
Broadband RF/IF & Digital Radio
Clocks & Timers
Data Converters
DLP & MEMS
High-Reliability
Interface
Logic
Power Management
Processors

- ARM Processors
- Digital Signal Processors (DSP)
- Microcontrollers (MCU)
- OMAP Applications Processors

Switches & Multiplexers
Temperature Sensors & Control ICs
Wireless Connectivity