Lab 5 - Building Programs with gMake

Goal

Welcome to the compulsory "Hello World!" lab exercise. Here we will begin our exploration of software programming tools. In this lab, you will:

- Use CCSv5 and GDB/GDBserver to build and debug a Linux application.
- Create a simple X86 makefile for building a specific program ("Hello World").
- Explore and analyze a more complex, generic makefile to be used throughout this workshop.
- Execute the "Hello world!" application on both the x86-based Linux host system and the ARM-based target system using Linux terminals.

Outline

Notes:

- Due to limited time in this TechDays session, we rearranged this lab to move the fun stuff (CCSv5 & GDB) earlier. If you have time, you can go on to explore making GNU make files in parts 05a/b and the Appendix.
- This lab was written to support 3 different boards. Today, though, we'll be using the OMAP3530 EVM.

Lab05d_standard_make

Part D – Analyzing TI's Standard Makefile

Since we will be using a standard makefile project in our debugging lab, it might be a good idea to try running the makefile from the command-line first. While this isn't required, we thought it may help when building code with this makefile from within the CCS/Eclipse GUI.

Note: The authors of this workshop have developed a "one size fits all" makefile for generating executables for the rest of the workshop. Of course, if you adopt this makefile back at work, you might have to change paths (in setpaths.mak), or alter some of the options (such as the targets or platforms) all depending on what you're doing. However, this solution is a pretty robust. (Later on, you may want to review the Appendix to this lab exercise for a thorough review of our makefiles.)

---

1. **Change to the `lab05d_standard_make/app` directory and list the files.**

   ```
   cd ~/labs/lab05d_standard_make/app
   ```

   You might notice that there are two makefiles in our directory:

   `makefile` is the PARENT
   `makefile_profile.mak` is the CHILD

   When you run *make*, `makefile` calls `makefile_profile.mak`. The main reason for having two files is to handle different profiles – *debug* and *release*. Otherwise, there would be a lot of duplicated code.

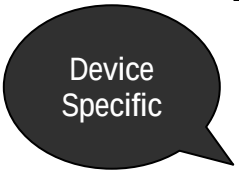
2. **Let's see some of the features of these make files by running them.**

   Let's start out easy and just make the *debug* profile:

   ```
   make debug
   ```

   Watch the screen. There is a LOT of information NOT being displayed. By designing the files the way we did, we tried to make the output look simple and uncluttered. We'll see how to turn on ALL the info in a few steps. List the contents of the directory again. Do you see **`app_debug.xv5T`**? If so, make worked. *(To repeat ourselves once again, in the next (optional) section – time permitting – you will open these files and browse their contents.)*

Note: Our executable programs use file extensions to differentiate between different target processors. While Linux doesn't require file extensions, this is a convenient way to allow several to co-exist, as well as just simply tell them apart.




```
.x86U      - Linux x86
.x470MV    - DM6446 ARM9 (using MontaVista toolchain)
.xv5T      - OMAP35x/AM35x (using Code Sourcery toolchain)
```

---

3. **Perform a "make clean" and observe the messages on the screen.**

4. **Using "help".**

   The authors built in some "help" information. Try:

   ```
   make help
   ```

   Peruse what just flashed before your eyes. These tips help you understand HOW to run this make file properly.

5. **Make "all".**

   Type:

   ```
   make all
   ```

   The all rule builds both the *release* and *debug* versions of the application. When gMake is done, you should see two executables: `app_debug.xv5T` and `app_release.xv5T`. You can't run these on an x86 PC, but next we will install them to the EVM so that we can run them to test if they are working properly.

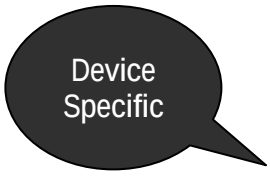
6. **Make "install".**

   Run a "make clean" first, then try:

   ```
   make install
   ```

   Executing just the *install* rule will automatically create the debug version of the application and install them to `/opt/workshop` directory on the EVM (which is `/home/user/targetfs/opt/workshop` within Ubuntu Linux). If you don't have a terminal open, open a terminal to the EVM using Tera Term. Log in as "root" and change to the `/opt/workshop` directory. Do you see the two executables?

7. **Run the debug executable.**



   Verify the debug executable works. You will need a "`./`" in front of the filename for the target board's *Linux* to recognize the filenames.

   ```
   ll  (lowercase LL – is an alias for ls –l)

   ./app_debug.xv5T
   ```

8. **Let's turn on some debug stuff…**

   The parent `makefile` allows you to specify debugging commands on the command line. Let's try two of the built-in "TELL ME EVERYTHING" switches.

   First, do a "make clean". Then, to allow gMake to echo each command it is asked to execute set "AT=" nothing on the command line:

   ```
   make clean
   make debug AT=
   ```

   Looks different, eh? Well, when (and if) you should NEED to view that information, this trick overrides the AT variable which is normally set to @.

   Do another "clean". There is also a "DUMP" switch that will output what each variable is set to (using gMake's *$(warning)* function), along with some other debug information. Trying it:

   ```
   make clean
   make debug DUMP=1
   ```

9. Check to verify that the *dependencies* rule is working correctly.

To verify the dependencies rule is working, first ensure everything is up-to-date by building with *debug* once again; then touch `app.h`, then try building again. If it runs the compiler the second time, it's working properly.

```
make debug
touch app.h
make debug
```

. Did gMake run *gcc* after `app.h` was touched (i.e. changed)? _____

. _____

Lab05e – Using Eclipse (CCSv5)

This last lab exercise explores using CCSv5 (i.e. Eclipse) for building and debugging our Linux applications. First, we'll install CCSv5; then set our project and remote debugging; then finally run/debug our program.

In the case of Linux applications, it's often convenient to use the GDB (Gnu DeBugger) protocol – running over Ethernet (TCP/IP) – for connecting between the host (CCSv5/Eclipse) and the target (Linux application running on the ARM). We'll find the gdb executable along with our build tools from Code Sourcery.

Here are a couple good references that you may want to refer to in the future:

CCSv5 wiki: http://processors.wiki.ti.com/index.php/Category:Code_Composer_Studio_v5

Linux Debug in CCSv5: http://processors.wiki.ti.com/index.php/Linux_Debug_in_CCSv5

CCSv5 Installation

Note: To save 10-15 minutes time, we have already installed CCSv5 for you.

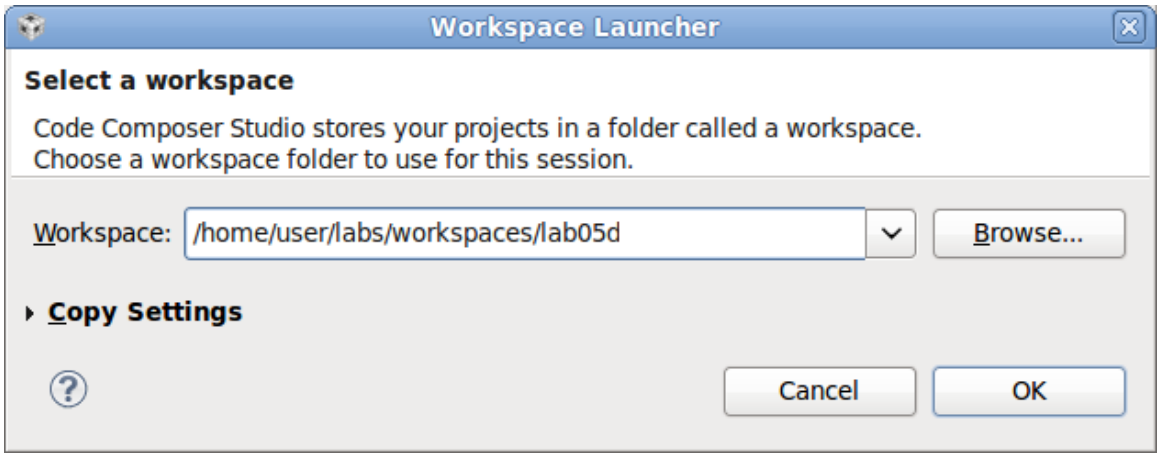
We've installed CCSv5 into the `/opt/ti` folder. While this wasn't the default location offered by the installer, we preferred to use one of Linux's standard program file locations.

Also, to keep the size down, we chose not to install the full Platinum edition, Rather, we only installed the C6000 DSP ISA support.

Create Project

10. **Start CCSv5 from the Ubuntu desktop.**

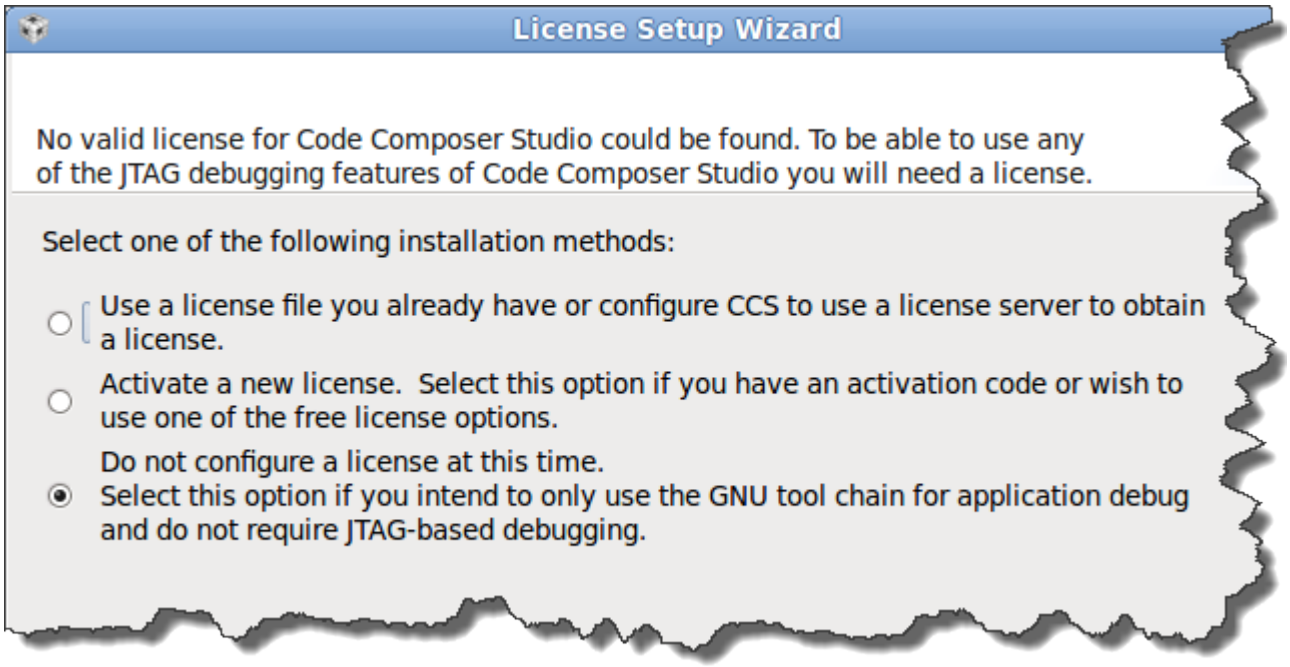
11. **Browse the following workspace.** *(Note, you may have to create this folder.)*



We chose the style followed by many Eclipse users, that is, to create a new workspace for each separate project; thus, only when projects are referenced by (or dependent upon) each other, would we put them in a common workspace.

12. **Close the *Welcome* window.**

    Note, we found that we needed to close the Welcome tab (“X”) in order to close this window.

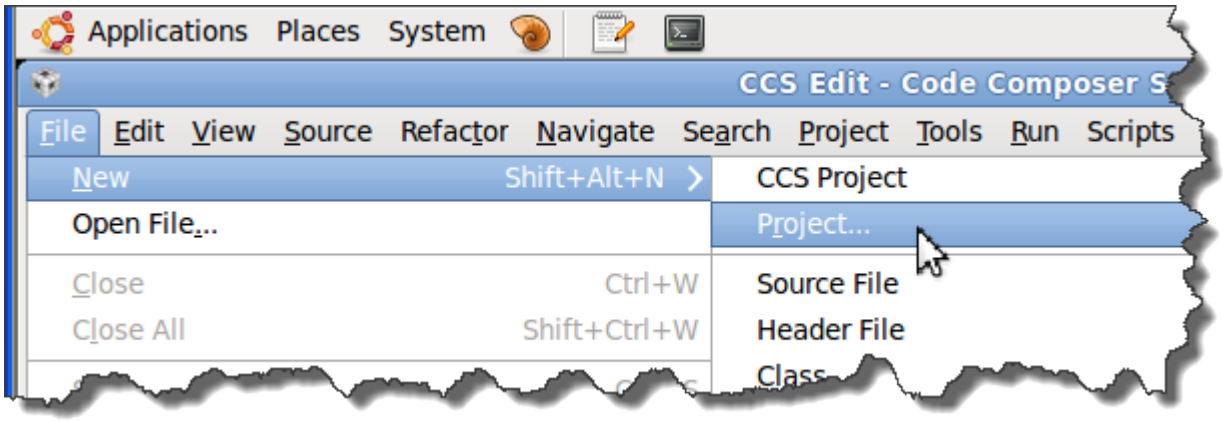


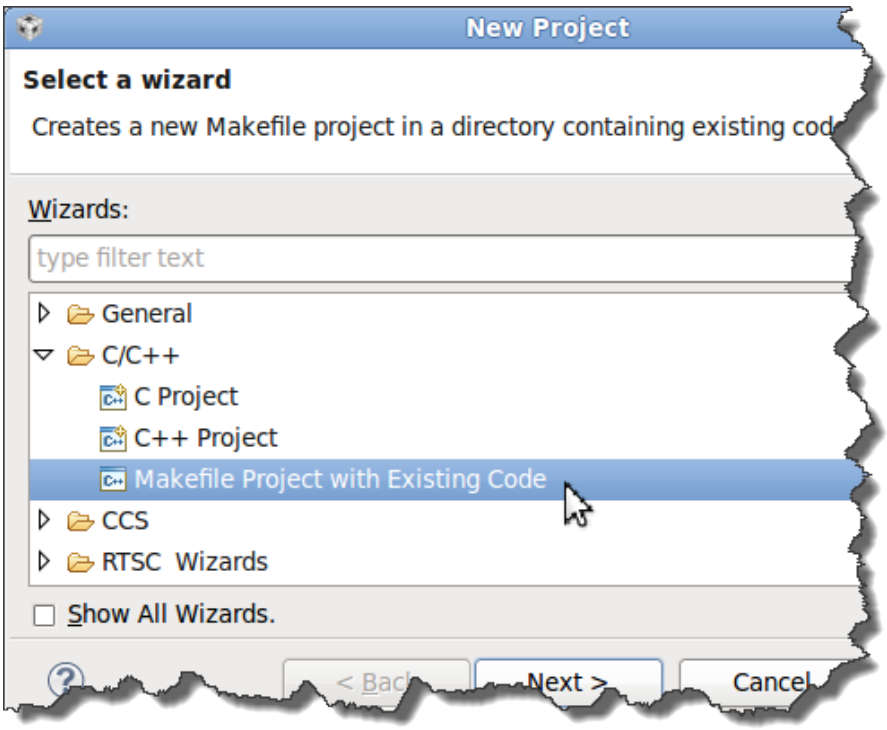
13. **Create a new project using our existing `lab05d` makefile.**

Eclipse provides many different types of projects. Most CCSv5 users choose one of two types:

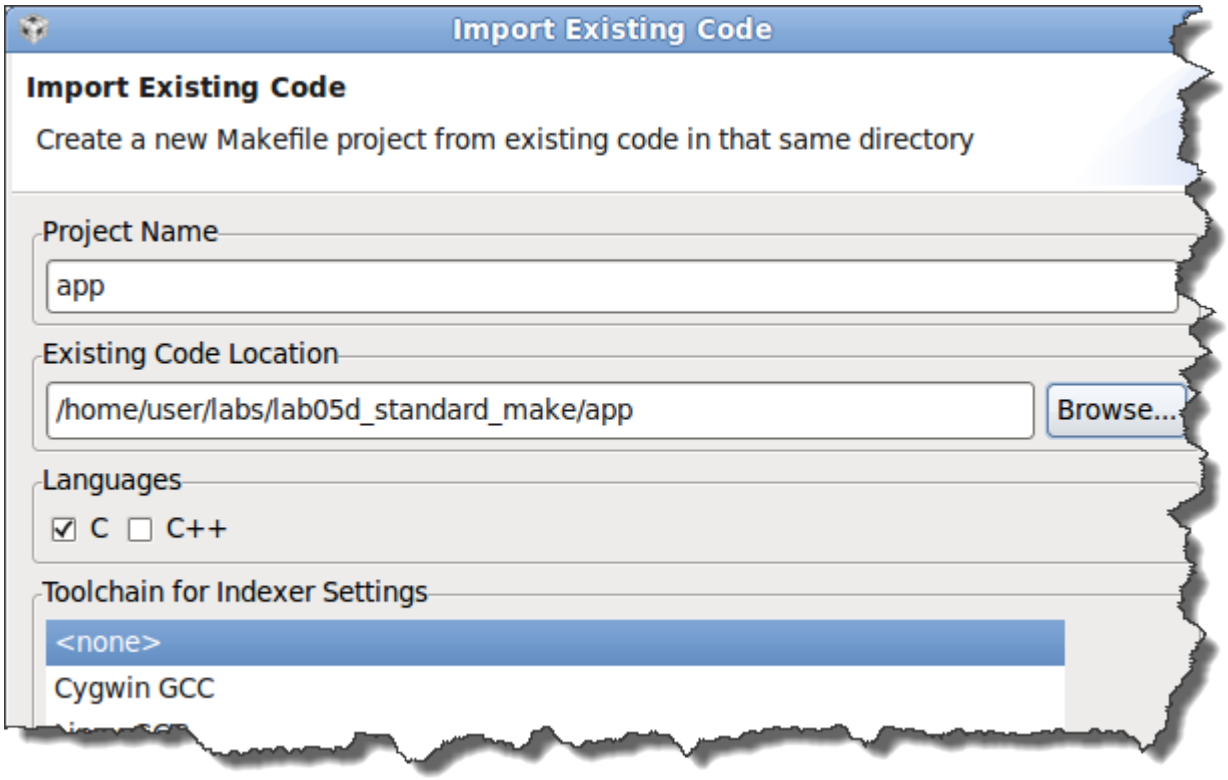
- **CCS Project** – uses Eclipse's managed make capability, which builds and maintains the make file for you as you add/subtract items and settings from the GUI
- → **Standard Make Project** – uses your own makefile; while this leaves the work of building and maintaining your own makefiles, it gives you absolute control over your builds

Since we already have our makefile already created, it's easier to use the standard make project.





14. **Import your existing makefile project.**

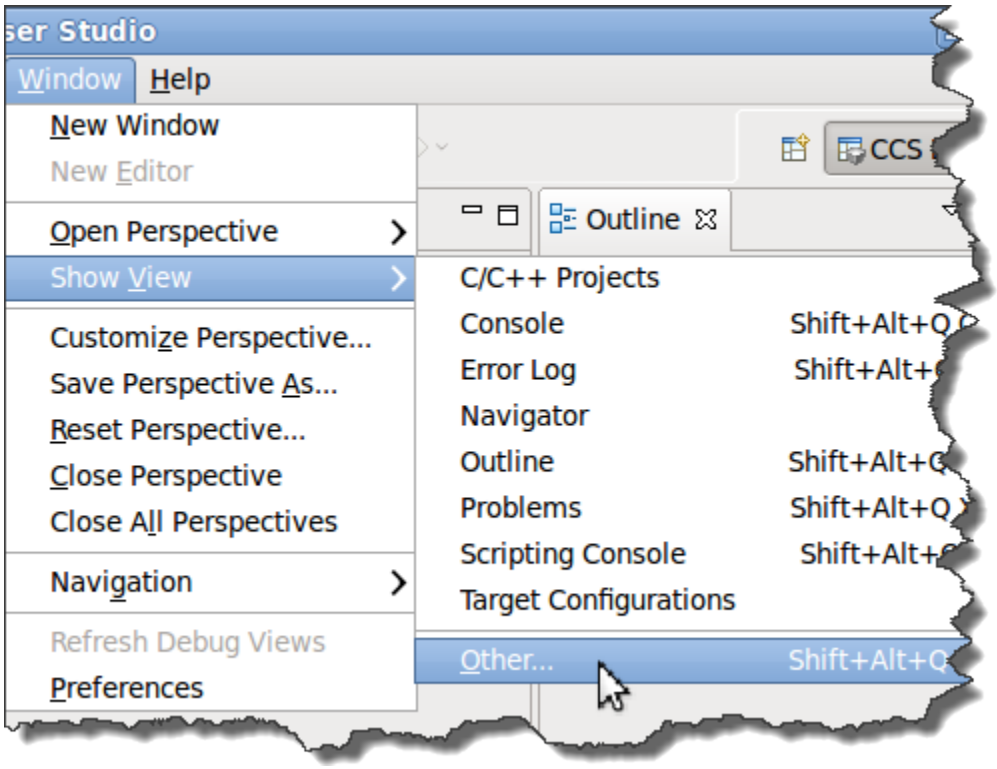


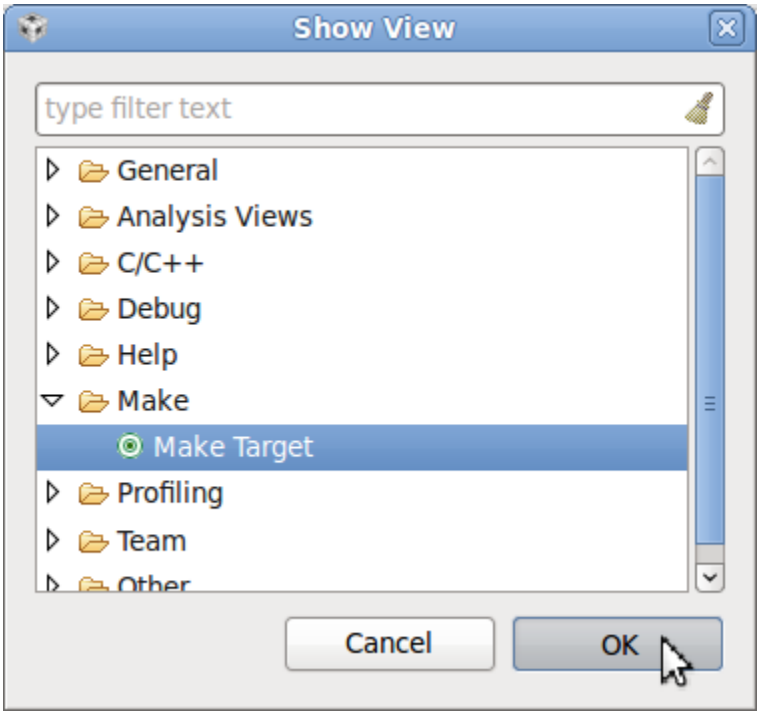
Click *Finish*, when done.

Show, Setup, and use the *Make Target* View

15. Show the Make Target view.

By default, the *Make Target* view is buried within Eclipse. This view allows to easily build any target within our makefile – although, (in the next step) we'll have to set it up first.



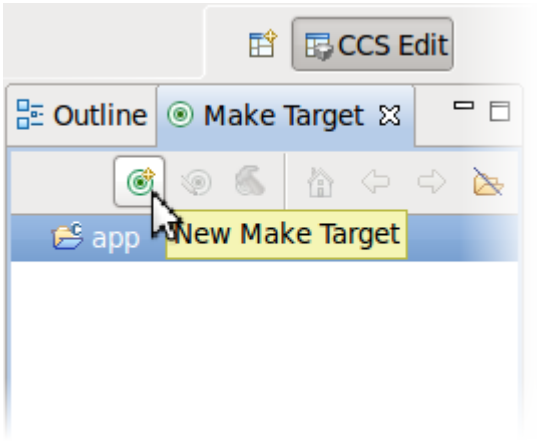


As a little challenge, figure out how to add the *Make Target* view to the *Show View* menu.

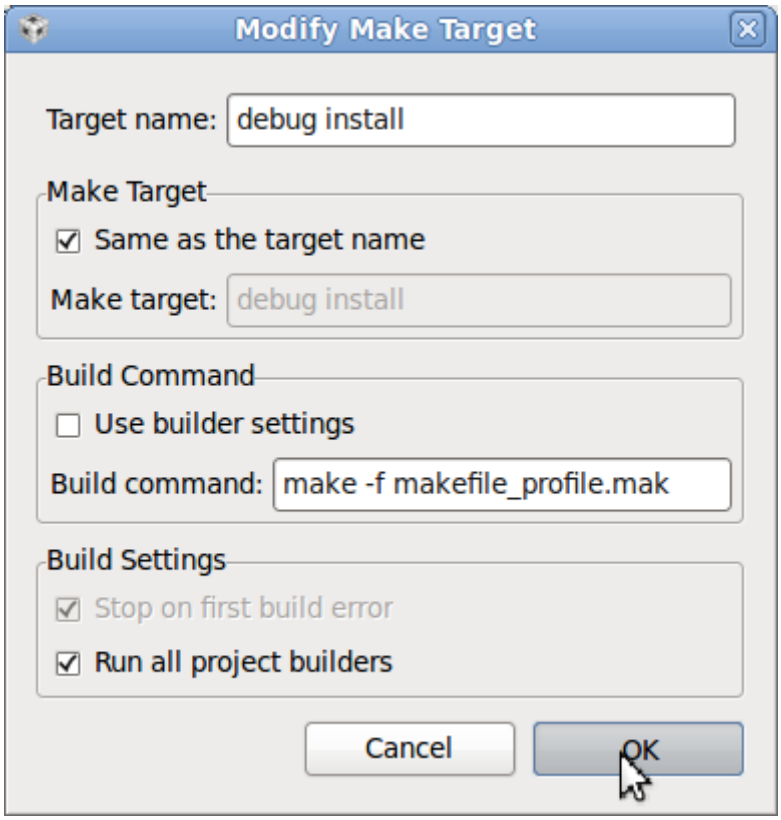
16. **Setup your Make Target view by adding our makefile targets.**

    Unfortunately, Eclipse doesn't provide an automatic means of parsing the makefile and presenting our build targets. Luckily, it isn't too difficult to add them manually.

    First, click the New Make Target button on the toolbar:



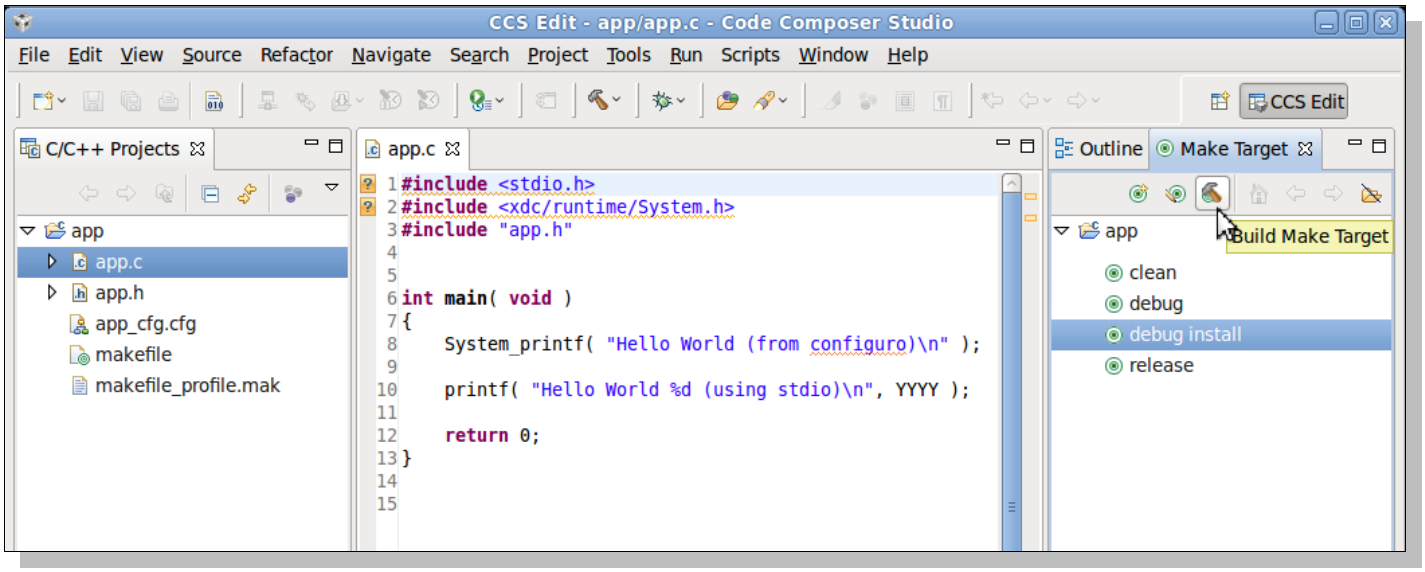
    Fill in the dialog as shown:



    Then, add some other needed targets:

    - clean
    - debug
    - release

17. **Build your program and install the executable.**



You should see the build feedback showing up in the Console window (not shown above), just like it did when running make from the Linux shell.

Start Program on ARM – waiting for GDB Debugger

18. **Go over to your ARM/Linux terminal (i.e. Tera Term) and navigate to our execute directory: `/opt/workshop`**

19. **Verify the IP address of your ARM running on the EVM.**

```
ifconfig
```

IP Address is: _____

20. **Start your program running in GDB debug mode.**

```
gdbserver <your EVM's IP address>:10000 <program to debug>
```

For example: **`gdbserver 192.168.1.122:10000 app_debug.xv5t`**

At this point, you shouldn't see much happen, yet.

```
root@omap3evm:/opt/workshop# gdbserver 192.168.1.122:10000 app_debug.xv5T
Process app_debug.xv5T created; pid = 1672
Listening on port 10000
```

You actually haven't run your program; rather, you've run the gdbserver program. The debug server will actually control your program as directed by a gdb client program – such as CCSv5. Next, we'll configure CCS to be a gdb client and talk over our IP address to port 10000.

Note: The number "10000" represents a networking port number. You can actually use any port number, but 10,000 is a common one since it's unlikely to be in use already.

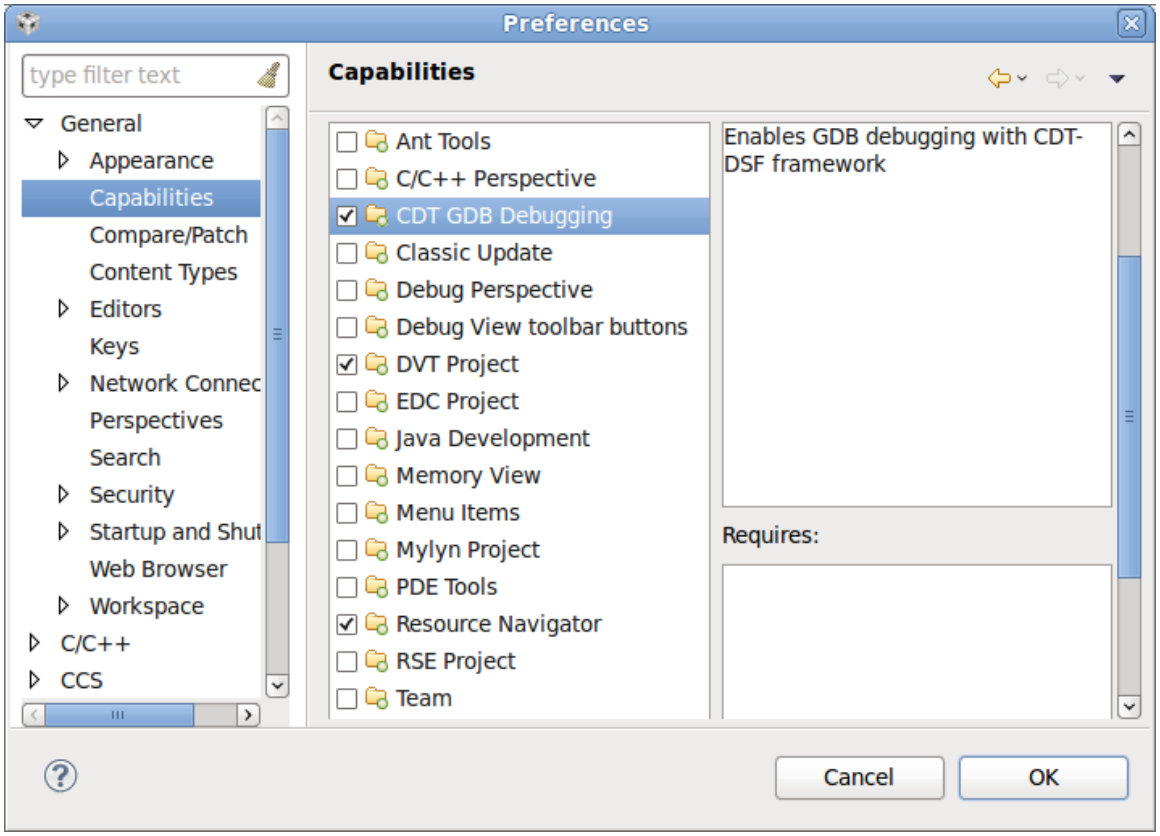
Setup CCCv5 For Remote GDB Debugging

IMPORTANT! By default CCS does not enable "C/C++ Application" configurations.

21. Enable the C/C++ Application configurations so that we can access GDB debugging from CCSv5.

Open the Capabilities tab in the CCSv5 Preferences dialog.

```
Window -> Preferences -> Capabilities
```



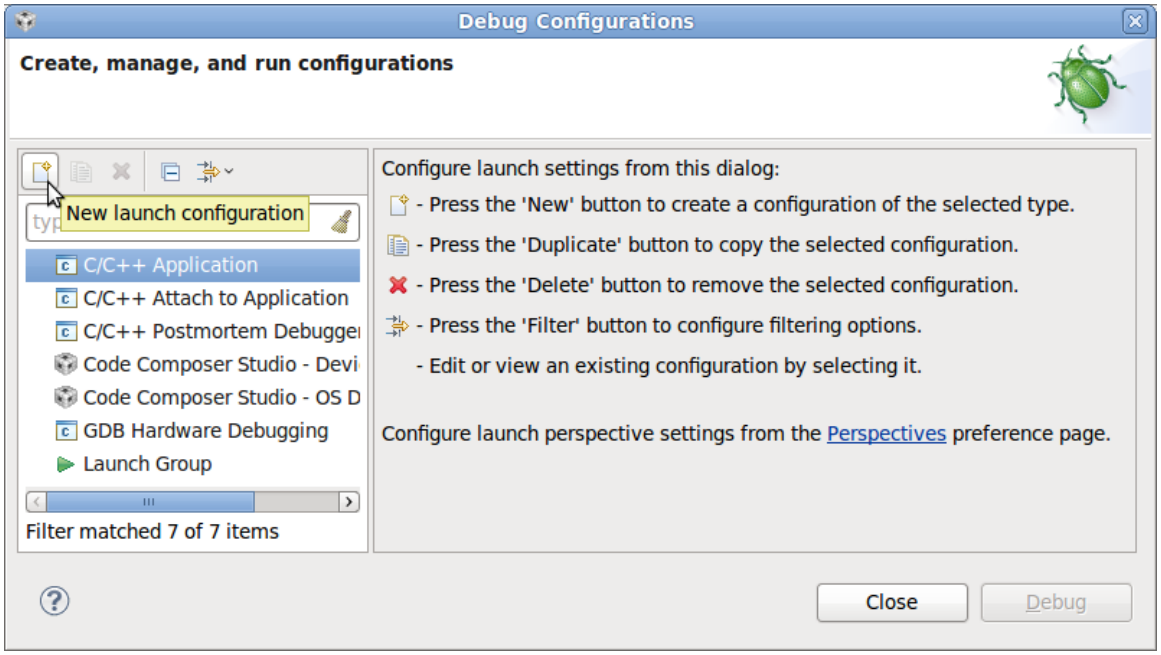
Enable *CDT GDB Debugging*, and then click *OK*.

22. Create a new C/C++ debug configuration.

Bring up the Debug Configurations dialog:

`Run → Debug Configurations`

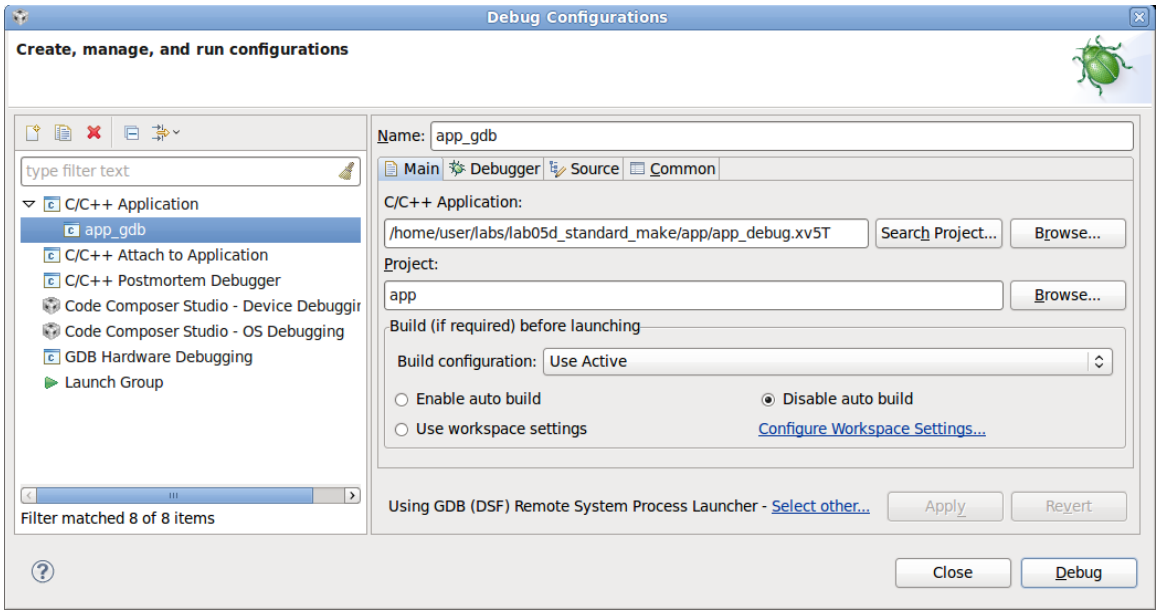
Select "C/C++ Application" and create a new configuration.



23. **Setup the configuration:**

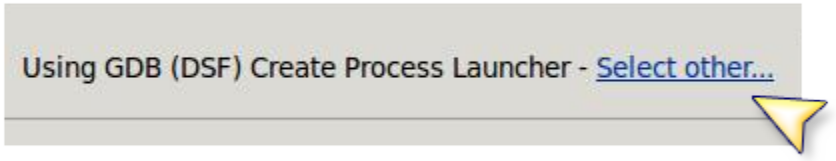
    Use the following settings:

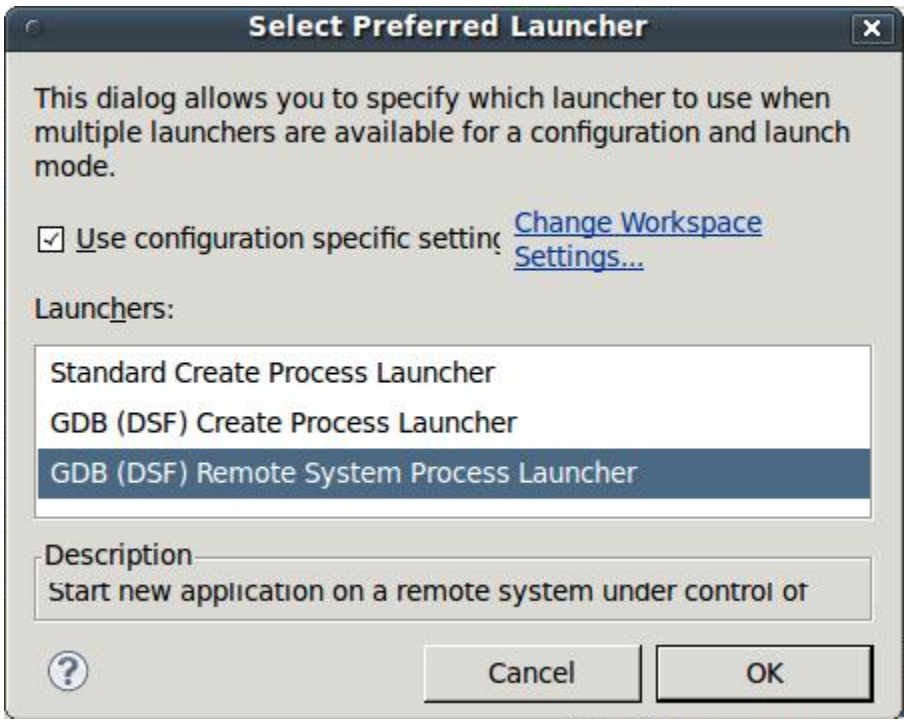
    - **Name:** app_gdb
    - **Project:** app
    - **C/C++ App:** /home/user/labs/lab05d_standard_make/app/app_debug.xv5T
    - **Disable** the auto-build option (as we are only interested in debugging our app).
    - *… but don't close the dialog, we're not done yet …*



24. Tell CCS that we want to use remote GDB debugging.

- Click the link "Select other..." to select a different launcher
- Select the "GDB (DSF) Remote System Process Launcher"
- Click OK

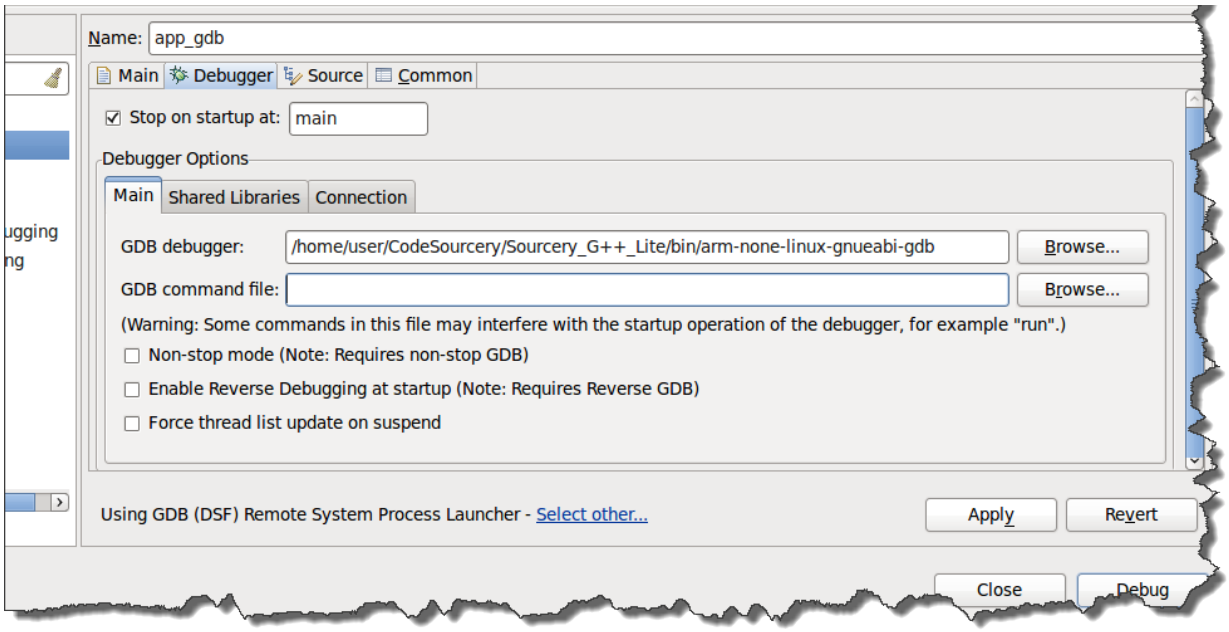




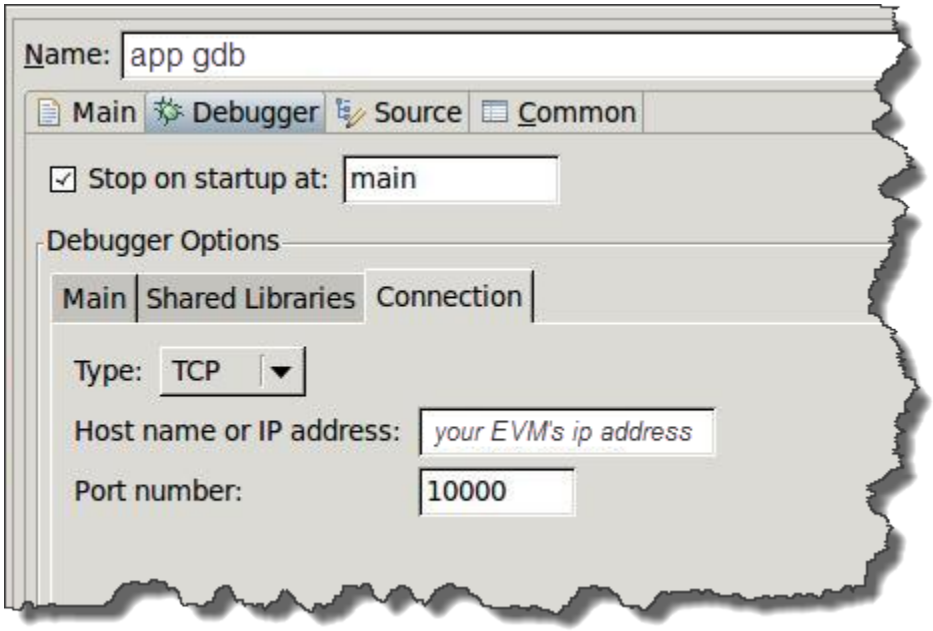
25. **On the *Debugger Main* tab, specify the GDB debugger.**

    We are using the GDB debugger from Code Sourcery, so browse for the correct gdb client executable.

    ```
    /home/user/CodeSourcery/Sourcery_G++_Lite/bin/arm-none-linux-gnueabi-gdb
    ```



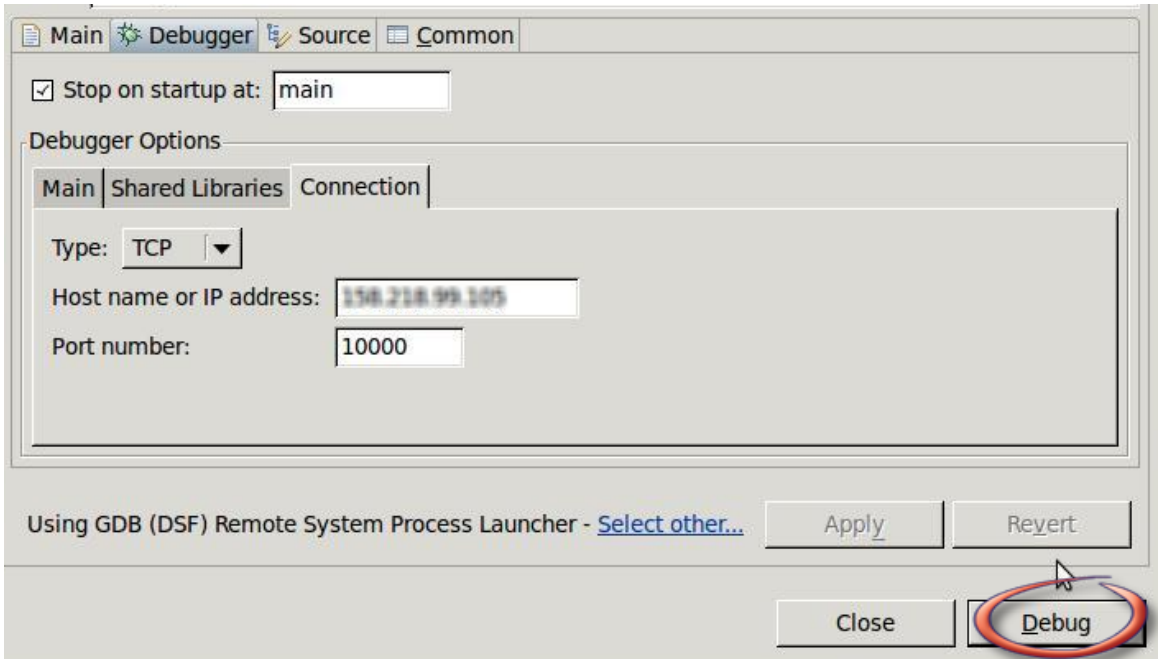
26. **On the *Debugger Connection* tab, specify the IP address and port of the GDB server running on the target.**



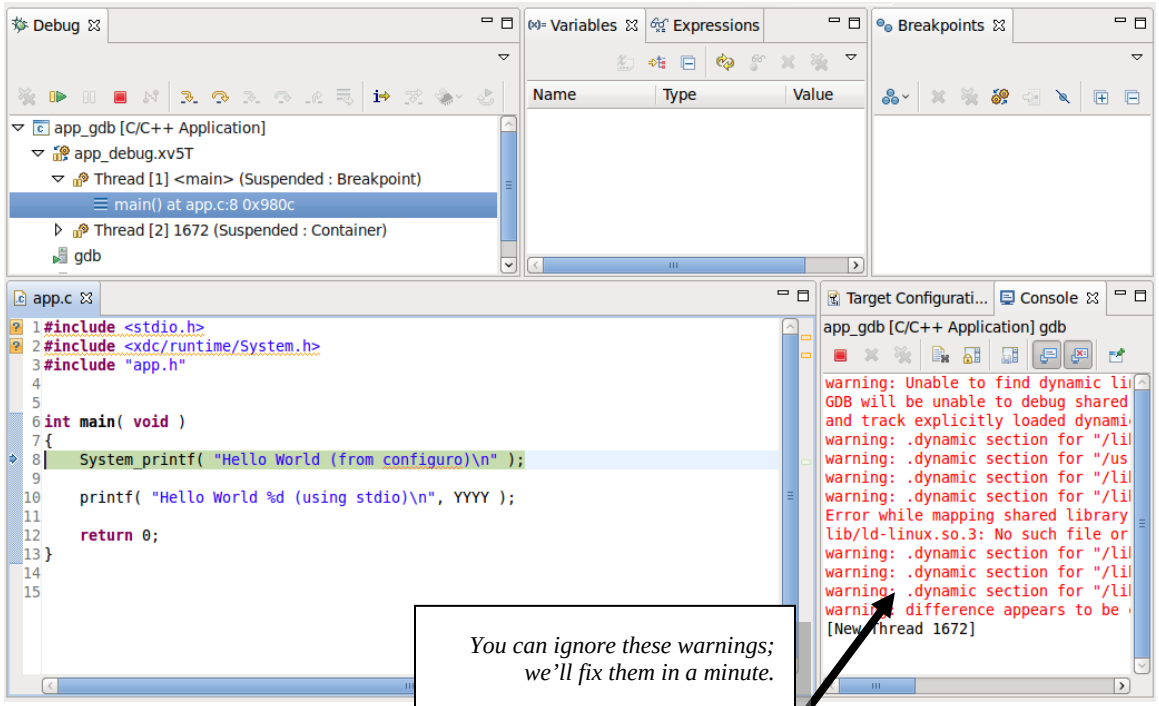
27. Launch the debug configuration by clicking the Debug button.

CCSv5 will launch the GDB debugger to connect to the GDB server.

After the connection is established, you can step, set breakpoints and view the memory, registers and variables of the "hello world" process running on the target.



After clicking *Debug*, the IDE will switch into the *Debug Perspective*. It will then load the program and execute until it reaches *main()*.



28. **Debug as you like – single-stepping & setting breakpoints.**

    While it's not a long program, this should give you an idea of how to get GDB working.

29. **Finally, check back in the Tera Term window to see the printf() results.**

```
root@omap3evm:/opt/workshop# gdbserver 192.168.1.122:10000 app_debug.xv5T
Process app_debug.xv5T created; pid = 1672
Listening on port 10000
Remote debugging from host 192.168.1.1
gdb: error initializing thread_db library: version mismatch between libthread_db and libpthread
gdb: error initializing thread_db library: version mismatch between libthread_db and libpthread
gdb: error initializing thread_db library: version mismatch between libthread_db and libpthread
gdb: error initializing thread_db library: version mismatch between libthread_db and libpthread
gdb: error initializing thread_db library: version mismatch between libthread_db and libpthread
gdb: error initializing thread_db library: version mismatch between libthread_db and libpthread
gdb: error initializing thread_db library: version mismatch between libthread_db and libpthread
Hello World 2010 (using stdio)
Hello World (from configuro)

Child exited with retcode = 0

Child exited with status 0
GDBserver exiting
root@omap3evm:/opt/workshop#
```

    Here, in Tera Term, you'll notice the same warning messages we saw in Eclipse. You'll also find the *printf()* statements. In this case, you'll find them inverted based upon fact that the System_printf() doesn't get written until this program exits.

Fix Library References

The warnings we saw when starting our debug session are due to Eclipse incorrectly referencing standard Linux libraries. By default, it's expecting to debug a native environment, rather than remotely debugging an ARM application. In the next few steps, we'll point Eclipse to the proper to the location where our ARM/Linux libraries reside.

30. **Create a new file called `.gdbinit` containing a reference to our target's filesystem.**

    You can perform this step from within CCSv5 or from the command line. It doesn't really matter. (We chose the latter method since it's easier to describe textually.)

    Similarly, it doesn't really matter where you place this file. You can do it right in the app directory, or you can place it at a common point, say the `/home/user/labs` directory. (In our case, and for no specific reason, we just put it in the app directory.)

```
cd /home/user/labs/lab05d_standard_make/app

gedit .gdbinit &
```

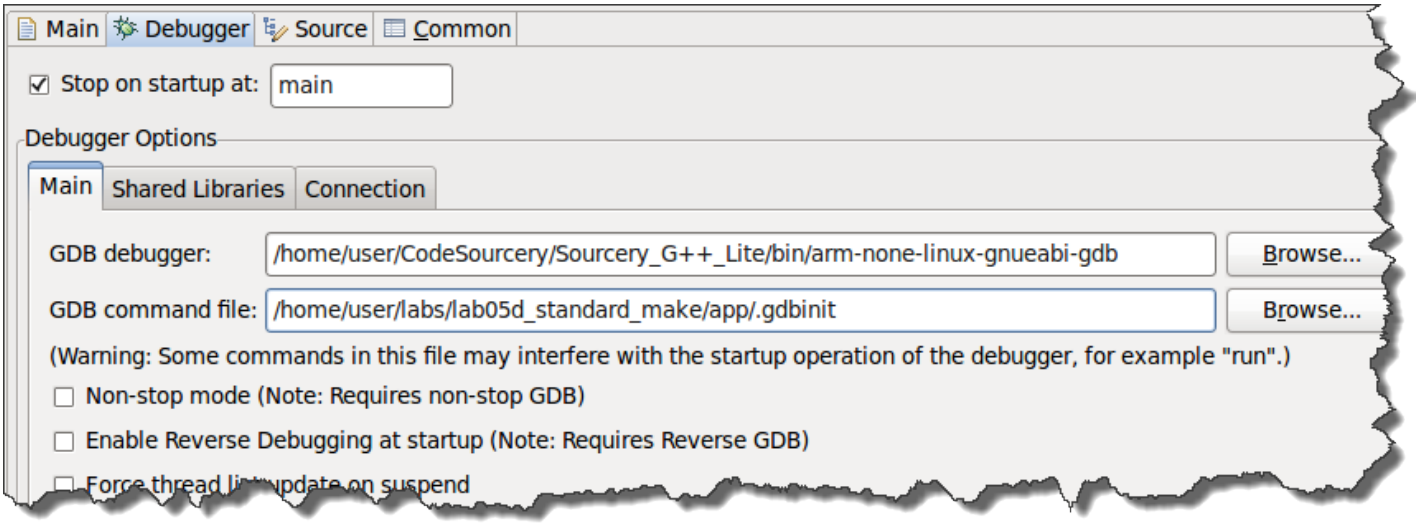
    Add the following text to the file, then save and close the file:

```
set solib-absolute-prefix /home/user/targetfs
```

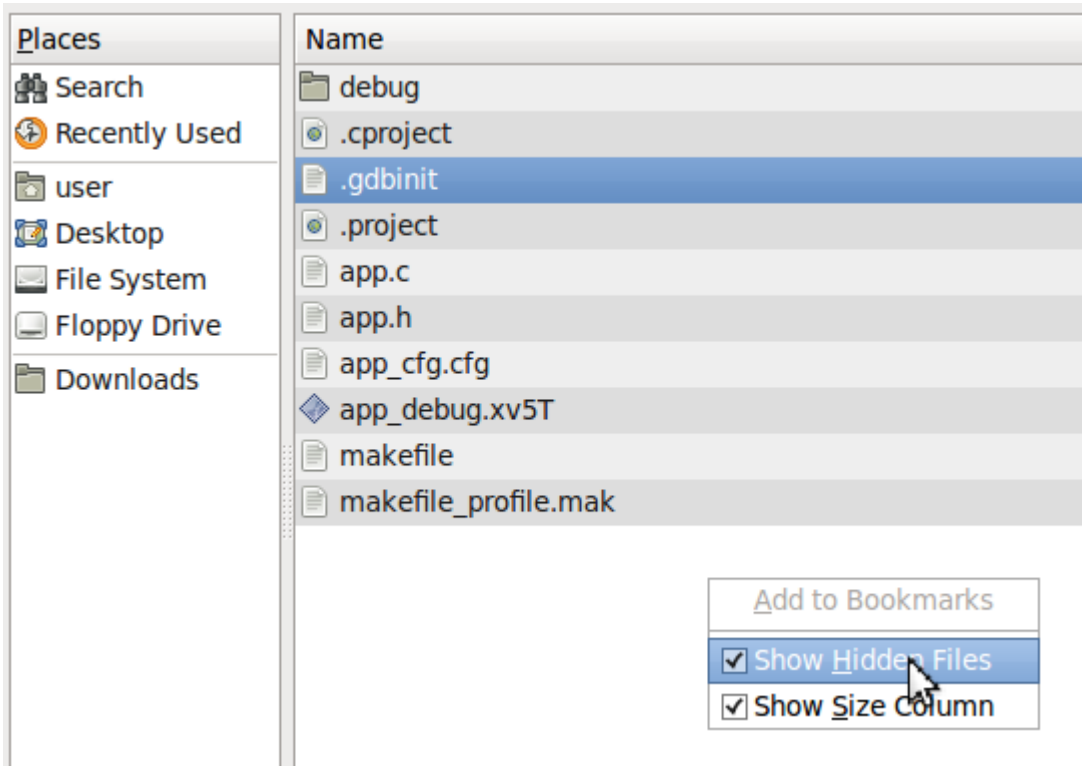
31. Add reference to `.gdbinit` to your *Debug Configuration*.

If your *Debug Configuration* is closed, reopen it from the *Run* menu.

Add the `.gdbinit` reference to the *GDB command file* (wherever you located it) to the dialog, as shown.



By the way, if you use the *Browse* button to enter the file location, you may need to right-click in the "open-file" dialog and chose to view invisible files.



32. Go ahead and *Apply*, then try *Debug* again…

(Optional) Lab05ab_basic_make

Big Picture

In part A of the lab, you will build your first basic makefile – basically turning command line execution into gMake rules. In Part B, you will increase the usability of your makefile by adding built-in variables and user-defined variables. This will provide you with a fundamental understanding of how makefiles work.

Procedure

Lab Prep – Examine the directory contents and app.c

33. Open a terminal in the Linux Host Computer.

Log into the Linux Host (i.e. desktop) Computer. Open a terminal window clicking on the "Terminal" toolbar icon.

You will begin in the `/home/user` directory (the home directory of the user named "user"), also represented by the tilde (~) symbol.

34. Locate the `labs` directory and list its contents.

Descend into to the `/home/user/labs` directory using the "cd" command. ("cd" is short for "change directory").

Use the "`ls`" (lower case "LS") command to ***list*** the contents of this directory:

```
ls
```

At any time, if you're curious about which directory you are in, use the Linux "pwd" command. This stands for "path working directory":

```
pwd
```

The `labs` directory is the working directory for the lab exercises and contains all the starter files needed for this workshop. (Note, solutions for each lab can be found at `/home/user/solutions`).

In addition to all of the lab folders, one of the additional files at this level is named `setpaths.mak`, which you will use later in this lab. `setpaths.mak` contains absolute paths for the locations of various tools and packages that will be used to build projects throughout the workshop. More on this later.

For this workshop, the proper file paths have already been configured for you. However, when you take your labs and solutions home to work on them further, you may need to modify `setpaths.mak` in order to build correctly on your system. (Note, the DVSDK uses the file named *Rules.make* for the same purpose as our *setpaths.mak*.)

35. Examine the contents of the `lab05abc_basic_make` directory.

```
cd ~/labs/lab05ab_basic_make
```

- or -

```
cd /home/user/labs/lab05ab_basic_make
```

- or, since you're probably already in the labs directory -

```
cd lab05ab_basic_make
```

List the contents of this directory. The `lab05ab_basic_make` folder contains only one directory, `/app`. (Later, as our lab exercises become more complex, some projects will have multiple directories at this level.)

36. Examine `app.c` in the `lab05ab_basic_make/app` directory.

Descend into the `app` directory. Examine the C source file `app.c` which prints the string "Hello World" to standard output.

```
cd app
gedit app.c
```

Part A – Using the Command Line and Creating a Simple Makefile

In this part, we will simply use the GNU compiler (`gcc`) from the command line to build the "Hello World" example and run it. Then, we'll place these commands into a basic makefile and run the makefile. In the next part, we'll use built-in and user-defined variables.

37. Build and run "Hello World" from the command line.

Make sure you are in `lab05ab_basic_make/app` folder.

To compile `app.c`, type the following command:

```
gcc –g –c app.c –o app.o
```

gcc	= GNU C compiler (command)
–g	= symbolic debug (compiler option)
–c	= (fill in answer below)
app.c	= file to compile (kind of "dependency" or "prerequisite")
-o	= output filename is next (compiler option)
app.o	= output file (the "target")

In the above gcc command, can you name the target, dependency and command?

- **Target = _____**
- **Dependency = _____**
- **Command = _____**

38. Use the "man" command to look up gcc.

To find the parameters for any standard C functions or Linux commands, you can use the "man" (short for "manual") command. Let's try it on gcc:

```
man gcc
```

What does the –c option (from step 37) tell the compiler to do?

_____

To quit the *man* page, type "q" at least once (depending on where you are in the page, you might need to type "q" multiple times).

39. Link the object file and produce the final executable.

Next, link the object file (app.o) to create the executable app.x86U:

```
gcc –g app.o –o app.x86U
```

Now run the executable:

```
./app.x86U
```

You should see "Hello World" displayed in the command window.

The extension used for the output file (.x86U) indicates we are building for the x86 (or host PC). In the future, we will build for the ARM target on the EVM and it will have a different extension (more on this later).

Note: For those of you who know Linux well, you can skip this explanation. For the rest …

> ./ before the name of an executable tells Linux to look for the program in the current directory.

We use this as it is the proper way to specify the path of the file to be run. Just in case you make a mistake and forget to include the ./, we added it to our Linux $PATH environment variable, so Linux will still be able to find your program.

---

40. "Clean" the existing executable (.x86U) and intermediate (.o) files.

Type the following to remove the files generated by the gcc commands you executed:

```
rm –rf app.x86U
```

```
rm –rf app.o
```

This removal of files mirrors what a "clean" macro or rule might do. We'll actually add a rule shortly to accomplish this in our makefile.

41. Examine "starter" `makefile`.

The current `makefile` in the `lab05ab_basic_make/app` directory simply contains comments and placeholders for the code you will write. Using your favorite editor, open the `makefile`. For example:

```
gedit makefile
```

42. Create rules for `app.x86U` and `app.o` in your `makefile`.

Remember, a rule is made up of a target, dependency(ies) and command(s). For example:

```
target : dependency
        CMD
```

Also note that the commands are tabbed over (at least one tab).

Create the rule for `app.o` in the area of the `makefile` with the header comments specifying the intermediate (.o) rule (as shown below). We'll help you with the rule for `app.o`, but `app.x86U` is up to you. If you get stuck, look back at the chapter material, ask the instructor for help or peek at the solution.

For `app.o`, type in the following rule. We will use the absolute path of gcc for now and later turn it into a variable:

```
# ------------------------------------------------
# ------ intermeditate object files rule (.o) -------
# ------------------------------------------------
app.o : app.c
     /usr/bin/gcc –g –c app.c –o app.o
```

app.o = target
app.c = dependency
/usr/bin/gcc -g … = command

43. Type in the rule for .x.

Next, type in the rule for `app.x86U` ABOVE the rule for `app.o` in the area specified for the (`.x`) rule. Make sure you use the –g compiler option in the `.x` rule.

44. Test your makefile.

Close makefile and type the following:

```
make
```

After running make, list the current directory.

```
ls
```

Do you see a new `app.x86U` executable? Run it:

```
./app.x86U
```

Do you see "Hello World"? If so, your rules work. Next, let's add a few more rules…

45. Open `makefile` in a different Linux process.

Stop. Before you open `makefile` again, try opening it in a different Linux process by typing in the following:

```
gedit makefile &
```

The "&" tells Linux to open the makefile in a separate process (window). When you edit a file, you can simply click Save, then click inside the terminal window and run it without having to re-open the makefile. Handy – and could save you some time.

46. Create a "clean" rule in your makefile

Whenever you run *gMake*, it will search and note the timestamps of the source files and executables and won't run if everything is up to date. So, it is common to create a "clean" rule that removes the intermediate and executable files prior to the next build.

In the `makefile` (underneath the comment header for "clean all"), add the following `.PHONY` rule for "clean" (these are the same commands you used earlier on the command line):

```
.PHONY : clean
clean  :
        rm –rf ___.x86U
        rm –rf _______
```

`.PHONY` tells *gMake* to NOT search for a file named "clean" because this is a phony target (i.e. it is not a file that needs to be searched for or created). In a large and complex makefile, this actually saves some compile time (plus, it is just good practice to use `.PHONY` when the target is not an actual file). The two files are the final executable and the intermediate object file.

47. Create an "all" rule in your `makefile`.

When *gMake* runs without any rules specified (i.e. you just type "`make`" on the command line), it will make (by default) the first rule in the `makefile`. Therefore, it is common to create an "all" rule that is placed first in the `makefile`. Our example only has one final target (`app.x86U`), so "all" doesn't make as much sense now. However, when we move to the `makefile` for the ARM target on the EVM, we'll have multiple targets to build and it will be more useful.

In the `makefile` (under the comment header for "make all"), add the following .PHONY rule for "all":

```
.PHONY : all
all : app.x86U
```

Close `makefile` and let's run it…

48. Run gMake to create the executable `app.x86U`.

On the command line, type in the following:

```
make
```

gMake will probably tell you that the files are "up to date" and there is nothing to do. So, you must run "clean" before you build again. Type:

```
make clean
```

and then:

```
make
```

or:

```
make all
```

gMake runs the first rule in the makefile which is the "all" rule. This should successfully build the `app.x86U` executable.

Note: *gMake* assumes the name of the make file is `makefile` or `Makefile`. *gMake* also looks for the FIRST makefile it finds. So, to be safe, you might want to capitalize Makefile because capital "M" comes before lower-case "m" alphabetically. You can also use a different name for the makefile – e.g. `my_makefile.mak`. In this case, you need to use the following command to "force" the use of a different make file name:

```
make –f my_makefile.mak
```

---

49. Run `app.x86U`.

You should see "Hello World" again. Ok, now that we have the simple makefile done, let's turn it up a few notches…

50. Review the different ways to run *gMake*.

As a review, you can run *gMake* in several ways:

`make`	(makes the first rule in the make file named makefile or Makefile)
`make <rule>`	(makes the rule specified with <rule>, e.g. "make clean")
`make –f my_makefile`	(forces the use of a make file named my_makefile)

Part B – Using Built-in and User-Defined Variables

In this part, we will add some user-defined variables and built-in variables to simplify and help the makefile more readable. You will also have a chance to build a "test' rule to help debug your makefile.

51. Add CC (user-defined variable) to your makefile.

Right now, our x86 `makefile` is "hard coded". Over the next few steps, we'll attempt to make it more generic. Variables make your code more readable and maintainable over time. With a large, complex makefile, you will only want to change variables in one spot vs. changing them everywhere in the code.

Add the following variable in the section of your `makefile` labeled "user-defined vars":

```
CC := $(LINUX86_GCC)
```

`CC` specifies the path and name of the compiler being used. Notice that `CC` is based on another variable named `LINUX86_GCC`. Where does this name come from? It comes from an include file named `path.mak`.

Open `path.mak` and view its contents. Notice the use of `LINUX86_GCC` variable and what it is set to.

Whenever you use a variable (like `CC`) in a rule, you must place it inside `$( )` for *gMake* to recognize it – for example, `$(CC)`.

After adding this variable, use it in the two rules (`.x` and `.o`). For example, the command for the `.x` rule changes from:

```
gcc –g app.o –o app.x86U
```

- to this -

```
$(CC) -g app.o –o app.x86U
```

52. Apply this same concept to the `.o` rule.

53. Add include for path.mak.

In the "include" area of the makefile, add the following statement:

```
-include ./path.mak
```

54. Test your makefile: clean, make and then run the executable.

55. **Add `CFLAGS` and `LINKER_FLAGS` variables to your makefile.**

    Add the following variables in the section of your `makefile` labeled "user-defined vars":

    ```
    CFLAGS := -g
    LINKER_FLAGS := -lstdc++
    ```

    `CFLAGS` specifies the compiler options – in this case, -g (symbolic debug). `LINKER_FLAGS` will tell the linker to include this standard library during build.
    (The example option `-lstd++` specifies the linker should include the standard C++ libraries.)

    Use these new variables in the .x and .o rules in your makefile.

56. **Test your makefile.**

57. **Add built-in variables to your .o rule.**

    As discussed in the chapter, *gMake* contains some built in variables for targets (`$@`), dependencies (`$^` or `$<`) and wildcards (`%`). Modify the `.o` rule to use these built-in variables.

    The `.o` rule changes from:

    ```
    app.o : app.c
            $(CC) $(CFLAGS) –c app.c –o app.o
    ```

    – to –

    ```
    %.o : %.c
            $(CC) $(CFLAGS) –c _____ -o _____
    ```

    Because we only have ONE dependency, ***use the $< to indicate the first dependency only***. Later on, if we add more dependencies, we might have to change this built-in symbol. `%` is a special type of *gMake* substitution for targets and dependencies. The `%.o` rule will not run unless a "`filename.o`" is a dependency to another rule (and, in our case, `app.o` is a dependency to the `.x` rule – so it works).

58. **Add built-in variables to your .x rule.**

    The .x rule changes from:

    ```
    app.x86U : app.o
            $(CC) $(CFLAGS) app.o -o app.x86U
    ```

    - to -

    ```
    app.x86U : app.o
            $(CC) $(CFLAGS) $(LINKER_FLAGS) _____ -o _____
    ```

59. **Don't forget to add the additional `LINKER_FLAGS` to the .x rule.**

60. **Test makefile.**

61. Add a comment to your .x rule.

Comments can be printed to standard I/O by using the `echo` command. In the `.x` rule, add a second command line as follows:

```
@echo; echo $@ successfully created; echo
```

The `@echo` command tells *gMake* to echo "nothing" and don't echo the word "echo". So, effectively, this is a line return (just like the echo at the end of the line). Because built-in variables are valid for the entire rule, we can use the `$@` to indicate the target name.

Test `makefile` and observe the echo commands. Did they work? As usual, you might need to run "make clean" before "make" so that *gMake* builds the executable.

62. Add "test" rule to help debug your makefile.

Near the bottom of `makefile`, you'll see a commented area named "basic debug for makefile". Add the following `.PHONY` rule beneath the comments:

```
.PHONY : test
test:
        @echo CC = $(CC)
```

This will echo the path and name of the compiler used. Try it. Does it work?

You can also add other echo statements for `CFLAGS` and `LINUX86_GCC`. This is a handy method to debug your makefile.

Close your makefile when finished.

Appendix – How We Built Our Workshop Makefiles

OPTIONAL – Analyzing the Details of the Makefiles

This optional lab takes you through some of the details of the two makefiles. At some point, if you decide to use these makefiles for your own builds, you'll need the information below. There are also some excellent references online to help you learn more about gMake.

Some great resources are:

http://www.gnu.org/software/make/manual/make.html

http://www.delorie.com/gnu/docs/make/make_toc.html

www.nso.edu/general/computing/TeX/local/texinfo/gmake/Top.html

And there are many more – just Google gmake and see what pops up.

Background for our makefiles - Introducing the parent and child makefiles.

We have actually developed a set of two makefiles (the parent – called `makefile`; and the child – called `makefile_profile.mak`). Here are just a few highlights of the overall capabilities of these makefiles:

- They can build using two different profiles: *debug* and *release*
- These makefiles build for the ARM target on the EVM. An install rule exists that automatically copies the executables to the proper directory on the EVM so that you can run via the Tera Term terminal.
- Full "clean" rule is provided.
- They handle dependencies (i.e. header files) from all .c files and any consumed packages.
- The parent takes the input from the command line and invokes the child with the proper profile and settings.
- There are also a few debug features built in to help find make script errors.
- The child does most all of the work - dependencies, configuro, .x and .o rules.

In this section (Part D), we only cover the use of these files. The next section (Part E – Challenge) encourages you to open up these files and learn more about their mechanics – but only if time permits.

Looking in the `lab05d_standard_make/app` directory and list the files.

Everything should look very similar – same `.c` and `.h` files, `app_cfg.cfg`, etc. However, there are two makefiles: `makefile` is the PARENT; `makefile_profile.mak` is the CHILD. When you run *make*, `makefile` calls `makefile_profile.mak`. The main reason for having two files is to handle different profiles – *debug* and *release*. Otherwise, there would be a ton of duplicated code.

```
cd ~/labs/lab05d_standard_make/app
```

1. **Browse the contents of the parent makefile: `lab05d_standard_make/app/makefile`.**

   Open `makefile` with a text editor:

   ```
   gedit makefile &
   ```

   We decided to use two makefiles to handle different profiles – these being "debug" or "release". If we only used one file to handle both profiles (and you could have more profiles than just two), you would end up repeating many of the rules and commands for each profile. So, instead of repeating this code over and over, we chose to let the parent (`makefile`) to call the child (`makefile_profile.mak`) with the appropriate profile setting. Thus, the parent makefile formats the user's request, then passes it onto the child makefile which contains the script to execute the detailed commands.

   You'll also notice that the parent makefile contains a lot of echoes/warnings to provide *help* as well as make gMake progress look clean and useful. You may or may not like the fancy syntax – and can change it to suit your needs if you apply it to your own projects back home.

   Let's take a brief look at the parent makefile (named `makefile`) – from top to bottom.

   A. The `AT` variable helps us turn on/off echoes from *gMake*. The default is to NOT echo all the commands that *gMake* spits out. You can leave this as is for a cleaner output – you can change it to "`AT :=` " in makefile – or, on the command line, use "`make debug AT = "` to change its value. As you go down into the file, you'll see how "`AT`" is used.

   B. gcc has implicit rules – such as compiling a `.c` file. Unfortunately, these default actions may not reflect the needs of cross-compiling (or anything project specific). We want to ignore them, otherwise we've found they occasionally override our explicit rules.

   C. gMake's filter function determines if you added "install" on the a command line, if so, then it's passed to the child makefile via the $(INSTALL) variable.

   D. Being the 1st rule found, the "all" rule runs if no target is specified on the command line.

   E. If no targets are specified along with "install", we build both *debug* and *release* profiles.

   F. Under the "Rules" heading, look at `debug` and `release`. We use the –f to call the child makefile (`makefile_profile.mak`), the INSTALL variable, and the profile (debug or release).

   G. The "`clean`" rule sends the child the "`clean`" goal along with the profile.

   H. The rest of the file contains the "`help`" rule – that tells you how to use this makefile.

makefile (i.e. "parent" makefile)

Step 1a

Debug variable AT. By default, it's set to "@".

It is used to prevent commands being echoed

Step 1b

Ignore make's implicit rules

Step 1c

If "install" is specified on the command line, then we set the INSTALL variable, otherwise leave it blank.

Step 1d

"all" rule – first in line makes it the default rule

Step 1e

If only "install" is specified as a target, then both debug and release are built (similar to call "all install".

Step 1f and g

Step 1g

```
# ------------------------------------------------------------------------------
# makefile                                                    (edited to fit this page)
#
# Use:
# - this is the PARENT makefile to makefile_profile.mak. You can specify any
#   child and run gMake with the proper options.
# ------------------------------------------------------------------------------

# AT: Used for debug purposes, it hides commands for a prettier output.
#     When debugging, you can set this to nothing on the make command line.
# ------------------------------------------------------------------
AT := @

# ------------------------------------------------------------------
# MAKE_OPTIONS:
#   - This variable is used to hold any command-line options that we might want to use for make.
#   - In our case, we have only added the option which forces make to ignore all the implicit
#     (i.e. built-in) rules. In our opinion, it's better to explicitly define each rule to be run.
# ------------------------------------------------------------------
MAKE_OPTIONS := --no-builtin-rules

# ---------    INSTALL: See description from 'help' below    ----------
# ------------------------------------------------------------------
ifeq ($(filter install,$(MAKECMDGOALS)),install)
  INSTALL := install
else
  INSTALL :=
endif

# -----------------              Rules
# ------------------------------------------
.PHONY  : all debug release clean install help

all     : debug release

ifeq ($(MAKECMDGOALS),install)
  install       : debug release
                  @echo "Install was called without other targets, so both 'debug' and 'release' were built"
else
  install       :
                  @echo
endif

debug   :
                  $(AT) make $(MAKE_OPTIONS) -f makefile_profile.mak $(INSTALL) PROFILE=debug | grep –v -F "directory"
                  @echo "Done building 'debug'"    ; echo
release :
                  $(AT) make $(MAKE_OPTIONS) -f makefile_profile.mak $(INSTALL) PROFILE=release | grep –v -F "directory"
                  @echo "Done building 'release'" ; echo
clean   :
                  @ echo "---- Cleaning up files for $(firstword $(MAKEFILE_LIST)) -----"
                  $(AT) make -f makefile_profile.mak clean PROFILE=debug   | grep -v -F make[1]
                  $(AT) make -f makefile_profile.mak clean PROFILE=release | grep -v -F make[1]
help    :
                  @echo "This makefile serves as a 'parent' (or master) makefile. That is, it calls another makefile
                  @echo "called 'makefile_profile.mak'. If the child makefile is called directly, it will build only
                  @echo "one profile (by default, it builds the debug profile). This parent makefile allows
                  @echo "you to easily build for multiple profiles with a single invocation.
                  @echo
                  @echo "The goals allowed by this makefile are:  all, debug, release, clean, install, help
                  @echo
                  @echo "   debug:  calls the child makefile with the "debug" profile "
                  @echo "release:  calls the child makefile with the "release" profile "
                  @echo "install:  adds the 'install' goal to the child makefile's target, then calls child. Install"
                  @echo "             install will only make the 'debug' profile and install it to the EVM"
                  @echo
                  @echo "To DUMP additional makefile variables, use 'DUMP=1' when you run make."
```

Note:

The makefile shown in print is the one included with the DM6446 lab exercises. There are a few minor differences in the OMAP/AM35 makefiles.

Overall, the parent simply handles the profiles and calls the child based on the goals listed when you invoke make. The child really does all the work to build the executables.

2. Open the child (`makefile_profile.mak`) which is called by the parent (`makefile`).

`Makefile_profile.mak` builds for the ARM9 target – however, other targets could easily be supported (with a little tweaking). The parent makefile passes the "PROFILE" (*release* and/or *debug*) and "INSTALL" variables to the child make file which performs the appropriate commands based on these parameters. All dependencies (e.g. header files) are handled by the dependency rule. The child uses Configuro to consume packages delivered by TI or 3rd parties (similar to how you wrote a previous part of this lab). All tools paths are specified in `setpaths.mak`, which is located in the `labs` directory (two levels above `app`).

In the following steps, we'll look at the main pieces of the child makefile to understand how it works. We'll do this chronologically from the top of the file to the bottom. Not every piece will be covered in detail, so referencing the links provided earlier may help you understand gMake even better.

```
gedit makefile_profile.mak &
```

3. "Early" Include file – `setpaths.mak`.

Near the top of `profile_makefile.mak`, you'll notice we included `setpaths.mak`. If you don't remember what is contained in this file, feel free to open it up and view its contents.

Files are included in two spots in this make script: early and late. In our case, we need the paths defined early on, otherwise a number of references would fail.

Conversely, if we include dependency (.d) files right away, that generates an error; therefore, we include these towards the end of the file.

4. User-defined variables – for the Compiler.

Under the comment banner"*User-defined Variables"*, you'll see the standard variable types that we used earlier, but notice that there are now two versions of compiler flags:

- debug (e.g. DEBUG_CFLAGS)
- release (e.g. RELEASE_CFLAGS)

Again, the parent passes the value of $(PROFILE) to the child at which point it's the child's responsibility is to build the executable program. You'll notice we need two sets of CFLAGS – one for each profile.

The standard CFLAGS and LINKER_FLAGS variables have been modified to appropriate flags needed to build ARM9 programs.

Note: If `makefile_profile.mak` was called without defining PROFILE, then it defaults to *debug*. A little later in this file we actually set PROFILE:=DEBUG to defines its default value.

---

makefile_profile.mak (i.e. ”child” makefile)

Step 3

Need to include our tool paths. They are defined in setpaths.mak and located such that this file is common for all labs.

Step 4

Along with the build flags for ARM gcc, we see there are two sets of flags associated with our “profile” choices (debug, and release).

```
# ******************************************************************************
#
# makefile_profile.mak
#
#
# Use:
# - Called by parent makefile named "makefile"
# - Can be called directly using gMake's -f option; refer to the syntax used
#   by the "parent" makefile to invoke this make file
# - Currently builds for ARM9 target, however other targets can be supported.
# - User can specify PROFILE (either debug
#   or release or all) when invoking the parent makefile
# - All dependencies (e.g. header files) are handled by the dependency rule
# - Uses Configuro to consume packages delivered by TI
# - All tools paths are specified in setpaths.mak located two levels above /app
#
# ******************************************************************************

# ******************************************************************************
#
#    (Early) Include files
#
# ******************************************************************************

# ------------------------------------------------------------------------
# setpaths.mak includes all absolute paths for DaVinci tools
#   and is located two levels above the /app directory.
# ------------------------------------------------------------------------
-include ../../setpaths.mak

# ******************************************************************************
#
#    User-defined vars
#
# ******************************************************************************

# ------------------------------------------------------------------------
# AT: - Used for debug, it hides commands for a prettier output
#     - When debugging, you may want to set this variable to nothing by
#       setting it to "" below, or on the command line
# ------------------------------------------------------------------------
AT := @

# ------------------------------------------------------------------------
# Location and build option flags for gcc build tools
#   - CS_GCC and CS_INSTALL_PATH are defined in setpaths.mak
#   - CC_ROOT is passed to configuro for building with gcc
#   - CC is used to invoke gcc compiler
#   - CFLAGS, LINKER_FLAGS are generic gcc build options
#   - DEBUG/RELEASE FLAGS are profile specific options
# ------------------------------------------------------------------------
CC      := $(CS_GCC)              # CC is used to invoke gcc compiler
CC_ROOT := $(CS_INSTALL_PATH)     # CC_ROOT is passed to configuro for building with gcc

CFLAGS       := -Wall -fno-strict-aliasing -march=armv5t -D_REENTRANT \
                -I$(CS_INSTALL_PATH)/include -I$(LLIB_INSTALL_DIR)/include -I$(PWD)
LINKER_FLAGS := -lpthread -L$(LLIB_INSTALL_DIR)/lib –lasound

DEBUG_CFLAGS   := -g -D_debug_ -D_DEBUG_
RELEASE_CFLAGS := -O2
```

5. **Creating arrays of C, object and dependency files.**

   Inspect the five lines of code that start with C_SRCS. The goal here is to create an array of object files and dependency files based on the existing C files in the current directory. So, we first create an array of .c files (C_SRCS) using gMake's wildcard function. Once we have this array, we can create a corresponding array of object files – C_OBJS – use two additional gMake functions (*subst* and *addprefix*). Similarly, we also use C_SRCS to create the array of dependency files – C_DEPS.

   Note, you'll see these variables being used further down in the child makefile.

6. **Inspecting the Configuro variables.**

   The next section should look familiar. You either wrote or copied this code in a previous part of this lab. To review, these are the variables that will be used in the Configuro rule later in the child makefile.

7. **Project specific variables.**

   Rather than hard-coding the program name, configuration filename, and profile, they have been created as variables. This should make it easier to adapt the makefile's for other programs/projects.

8. **Understanding PRECIOUS.**

   Scroll down a small amount and find the directive .PRECIOUS. This might be new to you, so let's explain it briefly. gMake, by default, deletes intermediate files unless you tell gMake not to. So, for instance if file1.c is used to build file2.o which is used in the final step to build file3.x470MV, then gMake may delete file2.o UNLESS you tell it not to. In our case, we don't want gMake to remove the C_OBJS array or the linker.cmd and compiler.opt files that Configuro creates. So, we use the .PRECIOUS directive to say "please DO NOT delete these files".

9. **Deleting implicit rules for object files.**

   gMake has implicit rules – i.e. if you don't tell it exactly what to do, it performs its own implicit rules. You could create a makefile with no rules, or rules with no commands, etc. So, we are just being a bit conservative here and telling gMake NOT to use any implicit rules for .o files. If you want to learn more about implicit rules, commands, etc., refer to the links provided earlier.

   Actually, we don't need this statement anymore since we've started calling make with the `--no-builtin-rules` option. Alas, we just didn't get around to eliminating this line from `makefile_profile.mak`.

makefile_profile.mak (cont'd 2)

Step 5

Array of source files to build; plus the object and dependency files derived from the C files in this directory

Step 6

Configuro related variables: 2 tool paths; search path; and, target/platform def's

Step 7

Name of program to build

Name of Configuro .cfg file

Default profile name

Step 8 is Precious

Step 9

FWIW, we like explicit rules

```
# ------------------------------------------------------------------------
# C_SRCS used to build two arrays:
#   - C_OBJS is used as dependencies for executable build rule
#   - C_DEPS is '-included' below; .d files are build in rule #3 below
#
# Three functions are used to create these arrays
#   - Wildcard
#   - Substitution
#   - Add prefix
# ------------------------------------------------------------------------
C_SRCS := $(wildcard *.c)

OBJS   := $(subst .c,.o,$(C_SRCS))
C_OBJS  = $(addprefix $(PROFILE)/,$(OBJS))

DEPS   := $(subst .c,.d,$(C_SRCS))
C_DEPS  = $(addprefix $(PROFILE)/,$(DEPS))

# ------------------------------------------------------------------------
# Configuro related variables
# ---------------------------
#  - XDCROOT is defined in setpaths.mak
#  - CONFIGURO is where the XDC configuration tool is located
#  - Configuro searches for packages (i.e. smart libraries) along the
#    path specified in XDCPATH; it is exported so that it's available
#    when Configuro runs
#  - Configuro requires that the TARGET and PLATFORM are specified
#  - Here are some additional target/platform choices
#      TARGET   := ti.targets.C64
#      PLATFORM := ti.platforms.evmEM6446
#      TARGET   := gnu.targets.Linux86
#      PLATFORM := host.platforms.PC
# ------------------------------------------------------------------------
XDCROOT := $(XDC_INSTALL_DIR)
CONFIGURO      := $(XDCROOT)/xs xdc.tools.configuro -b $(PWD)/../../config.bld

LAB_DIR        := $(PWD)/..
export XDCPATH := $(LAB_DIR);$(LAB_DIR)/../myDisplay/packages;$(DMAI_INSTALL_DIR)/packag

TARGET   := gnu.targets.arm.GCArmv5T
PLATFORM := ti.platforms.evm3530

# ------------------------------------------------------------------------
# Project related variables
# -------------------------
#   PROGNAME defines the name of the program to be built
#   CONFIG:  - defines the name of the configuration file
#            - the actual config file name would be $(CONFIG).cfg
#            - also defines the name of the folder Configuro outputs to
#   PROFILE: - defines which set of build flags to use (debug or release)
#            - output files are put into a $(PROFILE) subdirectory
#            - set to "debug" by default; override via the command line
# ------------------------------------------------------------------------
PROGNAME := app
CONFIG   := app_cfg
PROFILE  := DEBUG

# ------------------------------------------------
# ----- always keep these intermediate files -----
# ------------------------------------------------
.PRECIOUS : $(C_OBJS)
.PRECIOUS : $(PROFILE)/$(CONFIG)/linker.cmd $(PROFILE)/$(CONFIG)/compiler.opt

# ------------------------------------------------
# --- delete the implicit rules for object files --
# ------------------------------------------------
%.o : %.c
```

10. **Default rule.**

    Being the first rule listed in the file, *Default_Rule* becomes the, ahem, default rule. Notice, this rule depends upon the ARM executable program we really want to build.

11. **Build executable.**

    The next rule builds the final executable – either the DEBUG profile, the RELEASE profile. This part should look pretty familiar to you based on previous sections of this lab.

    Notice the use of PROFILE as a variable. If we're building both *debug* and *release*, we'll do this rule twice in order to build both executables. (That is, both can be built, but only by the parent `makefile` calling `makefile_profile.mak` for each profile.)

    Having to manage PROFILES (debug and release) is made much easier by using two makefiles. Otherwise, you have a lot of duplicate code in a single makefile. (Actually, our first attempt was to do it in one file – and it was VERY long – intimidating – so, we decided to have one makefile call another – and, in a way, it taught us the concept of multiple – recursive – makefiles.)

    As a side-note, we found it helpful to see a "count down" in the build output to the finish. What does that mean? In the echo statements, you'll see a "1. ----- …", "2. ---- …" etc, that provides an indication of how far gMake still has to go until it is finished. So, the last step – building the executable – is actually "1". The first step is actually "4". So, when you build using these makefiles, you'll see the echo statements reflect 4…3…2…1… and then it finishes. This is not necessary for the build – it just makes the information output to the stdout window easier to read.

12. **Object File Rule.**

    The .o rule should also look familiar. Nothing new here except for the PROFILE variable.

    The PROFILE variable here represents the subfolder we are placing our intermediate files into. This is done so that we don't overwrite our *debug* variables when building *release*, and vice-versa.

    The key to understanding this `target:dependency` rule is to *follow the %*:

    ```
    $(PROFILE)/%.o : %.c $(PROFILE)/$(CONFIG)/compiler.opt
    ```

    That is, just remember that **%** represents a substitution symbol. So, if I have a source file named:

    ```
    bar.c
    ```

    Then (when building for *debug*) I'll end up with a target object file named:

    ```
    DEBUG/bar.o
    ```

    While this might be obvious so many of you, it's a common question we get asked regarding this rule.

makefile_profile.mak (cont'd 3)

Step 10

Default_Rule is the default rule … wait do I hear an @echo here?

Step 11

An executable rule, similar to early parts of this lab.

We've only added *profile* (i.e. path) variables (and a few comments) to the .x rule

Step 12

Similar to the .x rule, we've added profile/path vars to the .o rule

```
# ****************************************************************************
#
#    Targets and Build Rules
#
# ****************************************************************************

# 1. Build Executable Rule  (.x)
# ------------------------------
#  - For reading convenience, we called this rule #1
#  - The actual ARM executable to be built
#  - Built using the object files compiled from all the C files in
#    the current directory
#  - linker.cmd is the other dependency, built by Configuro
# ----------------------------------------------------------------------
$(PROGNAME)_$(PROFILE).xv5T : $(C_OBJS) $(PROFILE)/$(CONFIG)/linker.cmd
        @echo; echo "1.  ----- Need to generate executable file: $@ "
        $(AT) $(CC) $(CFLAGS) $(LINKER_FLAGS) $^ -o $@
        @echo "          Successfully created executable : $@ " ; echo

# ----------------------------------------------------------------------
# 2. Object File Rule (.o)
# ------------------------
#  - This was called rule #2
#  - Pattern matching rule builds .o file from its associated .c file
#  - Since .o file is placed in $(PROFILE) directory, the rule includes
#    a command to make the directory, just in case it doesn't exist
#  - Unlike the TI DSP Compiler, gcc does not accept build options via
#    a file; therefore, the options created by Configuro (in .opt file)
#    must be included into the build command via the shell's 'cat' command
# ----------------------------------------------------------------------
$(PROFILE)/%.o : %.c $(PROFILE)/$(CONFIG)/compiler.opt
        @echo "2.  ----- Need to generate: $@ (due to: $(wordlist 1,1,$?) ...)"
        $(AT) mkdir -p $(dir $@)
        $(AT) $(CC) $(CFLAGS) $($(PROFILE)_CFLAGS) \
                    $(shell cat $(PROFILE)/$(CONFIG)/compiler.opt) -c $< -o $@
        @echo "Successfully created:  $@ "
```

13. Handling C File Dependencies.

This part of the makefile may be new to you. We discussed it in the chapter, but not in full detail.

This rule uses the compiler to create a dependency (**.d** for dependency) file which corresponds to each .c file in the current directory. What does the .d file contain? A list of dependencies (i.e. header files) referenced by the .c file.

These .d files helps gMake do what it's good at, trigger the rule to run if any of the dependent files are newer than the target. It is common to miss including header files as dependencies for .c targets; using the compile to generate this information is a great solution to the problem.

The `-MM` gcc option is used to tell the compiler to capture this dependency information, rather than compiling the file. We still provide it the same flags and files, though, just as if we were compiling the file.

In our rule, we pipe the outputs of the `gcc -MM` command into a file. We then format the compiler's output using a gMake macro (*format_d*). We adapted a set of commands – found on various gMake related websites – that reformat this list of dependency files into a gMake rule. For example, `app.o` depends on `app.h` (along with any other header file listed in `app.c`).

When gMake runs these rules, it checks the dates on the header files to see if any are newer than the corresponding .o file.

In the *format_d* macro (found near the bottom of the file), you'll see its command uses a string "reformatting" tool – **sed** – which stands for "stream editor". Sed is a convenient – albeit cryptic - way to process text strings.

14. Config Rule(s).

The Configuro rule should look familiar. Except for the PROFILE path, this should be nearly what you added to your makefile in a previous part of the lab to run Configuro; and thus, consume a package (e.g. for consuming the *system_printf()* function in `app.c`) .

The only other change we made was to alter the information output when running Configuro. We have made the Configuro output into a sort of quiet mode by piping them into a log file. If, on the other hand, you want to see this information, you can set DUMP=1 on the command line and all Configuro's verbosity will be displayed.

Finally, we added one last Configuro related command to prevent an error in the case where our specified .cfg file doesn't exist. The following command:

```
touch $(CONFIG).cfg
```

prevents this error condition. If Configuro attempts to run without a .cfg file, an error causes gMake to stop. So, when/if that case occurs, we create an empty config file using *touch*. This shouldn't hurt anything (unless you just forgot to provide the .cfg file), because Configuro fires up, sees that you haven't included any packaged content, and exits. Since we did not specify .cfg files as PRECIOUS, this temporary, intermediate file is deleted by gMake (as per its standard operating procedure).

makefile_profile.mak (cont'd 4)

Step 13

We're depending on this rule to make sure no changes are missed

Step 14

Configuro, Configuro, where art thou Configuro

The ifdef DUMP is used to 'quiet' its output

The touch command prevents an error for programs that don't need a .cfg file (i.e. aren't using RTSC packaged content)

```
# ------------------------------------------------------------------------
# 3. Dependency Rule (.d)
# ----------------------
#  - Called rule #3 since it runs between rules 2 and 4
#  - Created by the gcc compiler when using the -MM option
#  - Lists all files that the .c file depends upon; most often, these
#    are the header files #included into the .c file
#  - Once again, we make the subdirectory it will be written to, just
#    in case it doesn't already exist
#  - For ease of use, the output of the -MM option is piped into the
#    .d file, then formatted, and finally included (along with all
#    the .d files) into this make script
#  - We put the formatting commands into a make file macro, which is
#    found towards the end of this file
# ------------------------------------------------------------------------
$(PROFILE)/%.d : %.c $(PROFILE)/$(CONFIG)/compiler.opt
        @echo "3.  ----- Need to generate dep info for:        $< "
        @echo "          Generating dependency file   :        $@ "
        $(AT) mkdir -p $(PROFILE)
        $(AT) $(CC) -MM $(CFLAGS) $($(PROFILE)_CFLAGS) \
                        $(shell cat $(PROFILE)/$(CONFIG)/compiler.opt) $<  > $@

        @echo "Formatting dependency file: $@ "
        $(AT) $(call format_d ,$@,$(PROFILE)/)
        @echo "Dependency file successfully created: $@ " ; echo

# ------------------------------------------------------------------------
# 4. Configuro Rule (.cfg)
# ------------------------
#  - The TI configuro tool can read (i.e. consume) RTSC packages
#  - Many TI and 3rd Party libraries are packaged as Real Time Software
#    Components (RTSC) - which includes metadata along with the library
#  - To improve readability of this scripts feedback, the Configuro's
#    feedback is piped into a results log file
#  - In the case where no .cfg file exists, this script makes an empty
#    one using the shell's 'touch' command; in the case where this
#    occurs, gMake will delete the file when the build is complete as
#    is the case for all intermediate build files (note, we used the
#    precious command earlier to keep certain intermediate files from
#    being removed - this allows us to review them after the build)
# ------------------------------------------------------------------------
$(PROFILE)/%/linker.cmd $(PROFILE)/%/compiler.opt : %.cfg
        @echo "4. -- Starting Configuro for $^  (note, this may take a minute)"
    ifdef DUMP
        $(AT) $(CONFIGURO) -c $(CC_ROOT) -t $(TARGET) -p $(PLATFORM) \
                           -r $(PROFILE) -o $(PROFILE)/$(CONFIG) $<
    else
        $(AT) mkdir -p $(PROFILE)/$(CONFIG)
        $(AT) $(CONFIGURO) -c $(CC_ROOT) -t $(TARGET) -p $(PLATFORM) \
                           -r $(PROFILE) -o $(PROFILE)/$(CONFIG) $<  \
                                          > $(PROFILE)/$(CONFIG)_results.log
    endif
        @echo "Configuro has completed; its results are in $(CONFIG) " ; echo

# ------------------------------------------------------------------------
# The "no" .cfg rule
# ------------------
#  - This additional rule creates an empty config file if one doesn't
#    already exist
#  - See the Configuro rule comments above for more details
# ------------------------------------------------------------------------
%.cfg :
        $(AT) touch $(CONFIG).cfg
```

15. Build, clean and Install.

Nothing here should surprise you from our earlier discussion. Most of what we added here was to add echo's to provide a bit more feedback during build.

The only other thing we've done is to add these dependencies to our rule:

```
$(INSTALL_OSD_IMAGE) $(INSTALL_SERVER)
```

We don't need these right now, but they'll be important later:

- In Chapter 7, we'll need an image in our installation/execution directory (on the ARM) that will be used for the On-Screen Display graphics.
- In Chapter 11, we'll need the DSP Server copied over to the execution directory.

In both of these cases, we chose to add a variable to the makefile which copies over the image & server, respectively. Until these features are needed, we've just set the variables to nothing. Later, when we need them, we'll have you edit the variables with the appropriate file names/paths.

makefile_profile.mak (cont'd 5)

all, clean, install

A common set of phony rules.

What does .PHONY mean? Only that these target names don't represent real filenames. Phony just tells gMake not to go looking for any files named: all, clean, or install.

```
# ******************************************************************************
#
#    "Phony" Rules
#
# ******************************************************************************

# ------------------------------------------------------------------------
#  "all" Rule
# -----------
#   - Provided in case the a user calls the commonly found "all" target
#  - Called a Phony rule since the target (i.e. "all") doesn't exist
#    and shouldn't be searched for by gMake
# ------------------------------------------------------------------------
.PHONY  : all
all     : $(PROGNAME)_$(PROFILE).xv5T
          @echo ; echo "The target ($<) has been built."
          @echo


# ------------------------------------------------------------------------
#  "clean" Rule
# -------------
#  - Cleans all files associated with the $(PROFILE) specified above or
#    via the command line
#  - Cleans the associated files in the containing folder, as well as
#    the ARM executable files copied by the "install" rule
#  - EXEC_DIR is specified in the included 'setpaths.mak' file
#  - Called a Phony rule since the target (i.e. "clean") doesn't exist
#    and shouldn't be searched for by gMake
# ------------------------------------------------------------------------
.PHONY : clean
clean  :
          @echo ; echo "--------- Cleaning up files for $(PROFILE) -----"
          rm -rf $(PROFILE)
          rm -rf $(PROGNAME)_$(PROFILE).xv5T
          rm -rf $(EXEC_DIR)/$(PROGNAME)_$(PROFILE).xv5T
          rm -rf $(C_DEPS)
          rm -rf $(C_OBJS)
          @echo


# ------------------------------------------------------------------------
#  "install" Rule
# ---------------
#  - The install target is a common name for the rule used to copy the
#    executable file from the build directory, to the location it is
#    to be executed from
#  - Once again, a phony rule since we don't have an actual target file
#    named 'install' -- so, we don't want gMake searching for one
#  - This rule depends upon the ARM executable file (what we need to
#    copy), therefore, it is the rule's dependency
#  - We make the execute directory just in case it doesn't already
#    exist (otherwise we might get an error)
#  - EXEC_DIR is specified in the included 'setpaths.mak' file; in our
#    target system (i.e. the EVM board), we will use /opt/workshop as
#    the directory we'll run our programs from
# ------------------------------------------------------------------------
.PHONY  : install
install : $(PROGNAME)_$(PROFILE).xv5T $(INSTALL_OSD_IMAGE) $(INSTALL_SERVER)
          @echo
          @echo  "0. -- Install $(PROGNAME)_$(PROFILE).xv5T to 'Exec Dir' --"
          @echo  "          Execution Directory:  $(EXEC_DIR)"
          $(AT) mkdir -p $(EXEC_DIR)
          $(AT) cp    $^ $(EXEC_DIR)
          @echo  "          Install (i.e. copy) has completed" ; echo
```

16. Macro: format_d

As stated before, this macro reformats the list of dependency files (created by running the compiler with the –MM option) into gMake rules. This allows make to verify that the dependent files (i.e. header file) timestamps are not later than the .o files created from the .c files that reference them. (Whew, that's a mouthful.) In other words, when a header file gets modified, you want the object (.o) file to be rebuilt.

Here, as we've seen elsewhere, we use the DUMP variable to inject additional debugging information. If DUMP exists, then we embed a *$(warning)* function into each .d file; this warning shouts out whenever the .d file is read by gMake. You probably won't need this, but it helped us track down a bug or two.

17. Including C_DEPS.

We include our .d files at the end of our make script, rather than the beginning. If we included them at the same time that `setpaths.mak` was included, we would receive an error; this error happens because we are including the array of files specified by C_DEPS, but that variable wasn't defined before `setpaths.mak` was included. Therefore, we've put it at the end of our make file.

As we've seen elsewhere gMake supports *ifeq*/*endif* conditional statements. The conditional statement says, include all the .d files unless the MAKE GOALS include *clean*. (We don't need the dependency files when cleaning, as our *clean* rule doesn't delete source files.)

Since make will try to read the .d files on the first pass, before be build any of the targets, the first time this include is run will likely result in an error. We can tell make to ignore this error by using the "-" symbol.

```
include foo      # don't ignore an error
-include foo     # ignore an error if it occurs when running this command
```

An odd, but handy aspect of gMake is that when an *included* file is updated during its execution, it forces gMake to re-run the entire make script over from the beginning. So, even if we get an (ignored) error the first time we run this *include*, once the .d files are created (by our .d rule), the make file will be re-executed and our *include* should work this time around.

One last little item to point out. The command:

```
-include $(C_DEPS)
```

is run recursively. If you were to look back how C_DEPS was defined, you'll notice we used "=" rather than ":=". This tells make we want this to be a *recursive* variable. This include statement is a perfect example of why we want this. In most cases C_DEPS will hold a string of filenames, e.g. "`app.d foo.d … bar.d`". Due to the nature of recursive variables, our single include command will end up acting like:

```
-include app.d
-include foo.d
…
-include bar.d
```

Pretty darn handy, huh?

makefile_profile.mak (cont'd 6)

```
# ****************************************************************************
#
#    Macros
#
# ****************************************************************************
# format_d
# --------
#  - This macro is called by the Dependency (.d) file rule (rule #3)
#  - The macro copies the dependency information into a temp file,
#    then reformats the data via SED commands
#  - Two variations of the rule are provided
#     (a) If DUMP was specified on the command line (and thus exists),
#         then a warning command is embed into the top of the .d file;
#         this warning just lets us know when/if this .d file is read
#     (b) If DUMP doesn't exist, then we build the .d file without
#         the extra make file debug information
# ----------------------------------------------------------------------
ifdef DUMP
  define format_d
   @# echo " Formatting dependency file: $@ "
   @# echo " This macro has two parameters: "
   @# echo "   Dependency File (.d): $1     "
   @# echo "   Profile: $2                  "
   @mv -f $1 $1.tmp
   @echo '$$(warning --- Reading from included file: $1 ---)' > $1
   @sed -e 's|.*:|$2$*.o:|' < $1.tmp >> $1
   @rm -f $1.tmp
  endef
else
  define format_d
   @# echo " Formatting dependency file: $@ "
   @# echo " This macro has two parameters: "
   @# echo "   Dependency File (.d): $1     "
   @# echo "   Profile: $2                  "
   @mv -f $1 $1.tmp
   @sed -e 's|.*:|$2$*.o:|' < $1.tmp > $1
   @rm -f $1.tmp
  endef
endif


# ****************************************************************************
#
#    (Late) Include files
#
# ****************************************************************************
#  Include dependency files
# -------------------------
#  - Only include the dependency (.d) files if "clean" is not specified
#    as a target -- this avoids an unnecessary warning from gMake
#  - C_DEPS, which was created near the top of this script, includes a
#    .d file for every .c file in the project folder
#  - With C_DEPS being defined recursively via the "=" operator, this
#    command iterates over the entire array of .d files
# ----------------------------------------------------------------------
ifneq ($(filter clean,$(MAKECMDGOALS)),clean)
  -include $(C_DEPS)
endif
```

Step 16

If we sed it before, we'll say it again. We created this macro to encapsulate the formatting of the file dependency info spit out by gcc's –MM option.

Sure, we could've just put these lines of script straight into our .d rule, but: (1) it would have looked messier; and (2) we wouldn't have had the chance to try out gMake macros …

Step 17

Lately, we've been making some pretty mean .d files.

Seriously, here's how our .d files get included into our build.

If we're cleaning, we're not going to include them.

No worries if the .d files don't exist by the time we execute this statement. This is actually to be expected. So, by adding the "-" before include, we're just telling make to ignore any errors caused by our "–include"

makefile_profile.mak (cont'd 7)

```
# ****************************************************************************
#
#    Additional Debug Information
#
# ****************************************************************************
#  Prints out build & variable definitions
# ----------------------------------------
#  - While not exhaustive, these commands print out a number of
#    variables created by gMake, or within this script
#  - Can be useful information when debugging script errors
#  - As described in the 2nd warning below, set DUMP=1 on the command
#    line to have this debug info printed out for you
#  - The $(warning ) gMake function is used for this rule; this allows
#    almost anything to be printed out - in our case, variables
# ----------------------------------------------------------------------

ifdef DUMP
  $(warning To view build commands, invoke make with argument 'AT= ')
  $(warning To view build variables, invoke make with 'DUMP=1')

  $(warning Source Files: $(C_SRCS))
  $(warning Object Files: $(C_OBJS))
  $(warning Depend Files: $(C_DEPS))

  $(warning Base program name : $(PROGNAME))
  $(warning Configuration file: $(CONFIG))
  $(warning Make Goals        : $(MAKECMDGOALS))

  $(warning Xdcpath :  $(XDCPATH))
  $(warning Target  :  $(TARGET))
  $(warning Platform:  $(PLATFORM))
endif
```

18. Print out build information.

In the last part of the child make file, you'll see a bunch of *$(warning)* statements. This is a handy way to print out some information on gMake variables, which could make debugging make easier. Looking at the file, you'll see these warnings will only show up if you have "DUMP=1" on the command line. (Alternatively, you could add the DUMP variable to the make file itself, but since we shouldn't need to debug this file anymore, defaulting to off is probably better.)