portos

The RTOS Revolution

Interface to TI BIOS

Softwave Wireless

http://www.portos.org

© 2003-2009 by Softwave Wireless

All rights reserved. No part of this manual may be reproduced in any form or by any means without prior agreement and written consent from Softwave Wireless.

Trademarks

Portos is a registered trademark of Softwave Wireless.

All other product and company names are trademarks or registered trademarks of their respective holders.

Contents

1 INTRODUCTION 4

2 CODE COMPOSER STUDIO PROJECT 5

3 DEBUGGING 8

4 CONFIGURATION 10

5 PRECOMPILED LIBRARIES 11

1 Introduction

For an introduction to Portos, please refer to *Portos User and Reference Guide*. This document is specific to the interface to Texas Instruments BIOS (TI BIOS).

Portos can be used either as a stand-alone RTOS or as an add-on library that complements an existing RTOS. Currently, the only available version of Portos is an add-on library that complements the TI BIOS. When using Portos as an add-on library to TI BIOS, you benefit from the features of both operating systems. Another important advantage is that the Portos library is (nearly) automatically ported to all DSPs that are supported by TI BIOS, and hence the hours of testing Portos on one DSP are leveraged by all other DSPs.

There are disadvantages though: the code size is not minimal since Portos as a standalone OS is much smaller. And the same can be said for speed of execution: stand-alone Portos runs faster. However, in most cases, these disadvantages are not significant.

In order for Portos to interface to the TI BIOS, your project must, of course, use the TI BIOS. In the current version of Portos for TI BIOS, the priority functions run in the same *virtual task* where the Software and Hardware Interrupts (SWI and HWI) run. In other words, the priority functions *share the same stack* as the one used by software and hardware interrupts. This stack should be increased in size if you have a large number of priority functions. If this stack is in the DSP's internal RAM and has size limitations, then you may not be able to define too many priority functions. A future version of Portos with supertasks will solve this issue. The regular tasks and SWI of the TI BIOS can still be used, but in most situations priority functions offer a simpler and more modular alternative.

Important note: the priority functions run at the SWI level. Therefore, from within priority functions it is not possible to call TI BIOS functions that are only callable from task level (e.g. TSK_create). A future version with supertasks will solve this issue.

If you have an existing project that uses tasks and SWI, you can simply start adding Portos instructions to that same project. Portos is fully compatible with tasks and SWI. Keep a backup of the old project in case you change your mind.

2 Code Composer Studio Project

If you are using an external makefile, you don't need to read this section. Refer to *Portos User and Reference Guide* for how to call the Portos preprocessor from a makefile.

If you are using a CCS project without external makefile then the job is not too easy because currently the CCS does not provide a simple GUI field to call the Portos preprocessor after the C preprocessor but before the C compiler. Hopefully the CCS will provide the necessary mechanism soon, and this (annoying) section will become obsolete.

For this reason, we created an additional tool to help you solve the issue. The tool does the job nearly automatically. You need to add the hooks to the tools, and every now and then make sure everything is properly working.

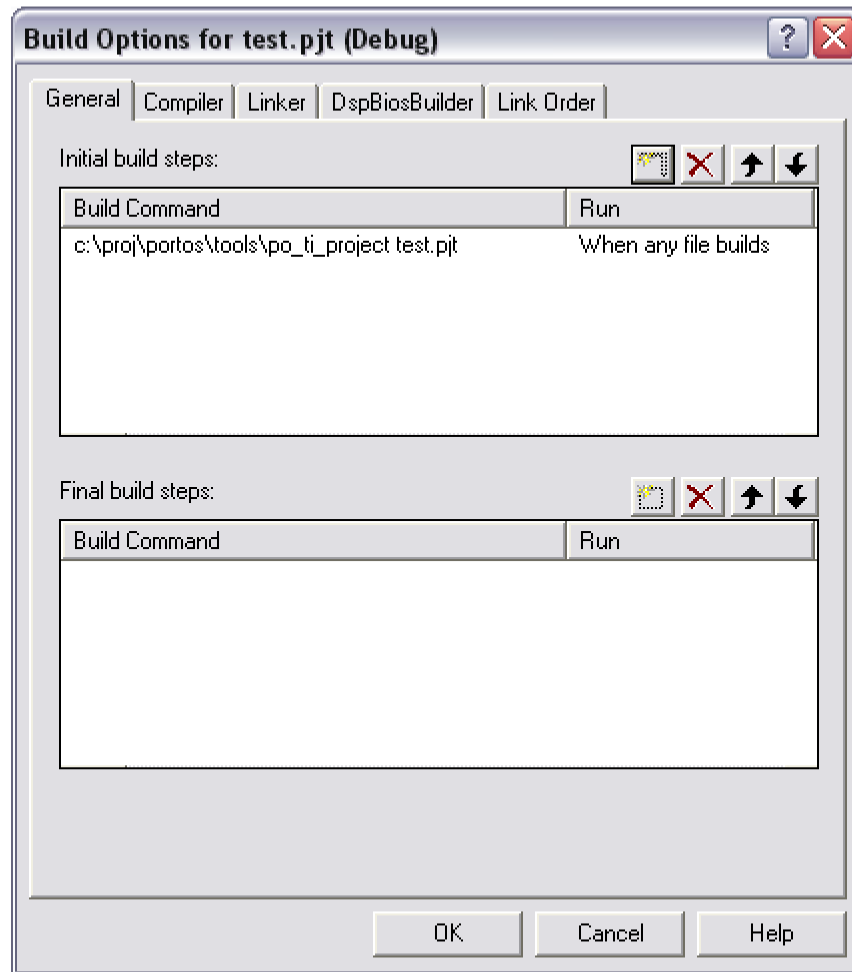
The tool is **`po_ti_project.exe`**. It is located under **`portos/bin`**. It automatically adds special instructions to a CCS project file in order to take care of calling the Portos preprocessor.

Do not call this tool directly but add the following instruction to your Build Options (cf. figure below). The instruction should be added as an *initial build step*, and it should be added for each configuration (e.g. Debug and Release):

```
<portos_path>\portos\bin\po_ti_project  myproject.pjt
```

where **`myproject.pjt`** is the project name. If you change the project name you have to modify it here too. In this example, **`<portos_path>`** is the path to the Portos directory.

The above instruction can be added in CCS as shown in the figure below. Open the project's **Build Options → General tab → Initial build steps → Build Command**. You can set the Run flag to either "When any file builds" or "Always". The second option is safer because CCS is sometimes unable to determine that certain flags changed and files need to be recompiled. But it will result in calling **`po_ti_project.exe`** too often.



After you add this instruction, every time you build or compile files, the tool **`po_ti_project.exe`** is called.

IMPORTANT NOTE: the tool assumes that you save the project file each time you make any modification to the project.

Make sure you save the project file every time you make some change to the project, such as modify compiler options or add new files. The tool **`po_ti_project.exe`** then reads the project file and it figures out whether there are changes and whether it should append new instructions in order ensure that the Portos preprocessor is called at the right time (as if using an external makefile). The tool will keep reminding you to save the project file.

More precisely, the tool **`po_ti_project.exe`** performs the following steps:

- Find all **`.c`** and **`.cpp`** source files that are new to the project.

- For each new file, create custom build instructions that call a special batch file (e.g. **`myproject.pjt.po.bat`**).

- Locate the CPU family name and obtain from the registry the appropriate compiler and its path.
- Locate the compiler options for each configuration (e.g. Debug, Release, etc).
- Create the batch file (e.g. **`myproject.pjt.po.bat`**) that acts like a makefile and ensures that the Portos preprocessor is called at the right time.

If one (or more) of your files already uses custom build instructions then this particular file needs extra attention. The tool will report a warning for this file and will not attempt to modify your custom build instructions. Then you will need to find a way to call the Portos preprocessor for this particular file. You can get ideas from the kind of instructions that **`po_ti_project.exe`** adds to the project file, and from the specially created batch file. In particular, look at the end of the created batch file, e.g. **`myproject.pjt.po.bat`**, and you will see that it can call your own program and arguments, **`%5 %6 %7 %8 %9`**, if you append them to the custom build instruction.

When you add new files to the project, the **`po_ti_project.exe`** tool has to make modifications to your project file. CCS will notice that the project file changed and will suggest that you reload it. You should reload it, reselect the proper configuration (e.g. Debug or Release) and then you may want to click Build again in order to build what has been modified (quite likely the previous build finished with errors).

IMPORTANT NOTE: the first time you create a project, you may have to click build several times before things go right. Because the tool **`po_ti_project.exe`** will modify the project but the CCS will use an old version… until it reloads the latest version.

When you change the build options, for example, sometimes CCS fails to recompile certain files. You may need to click the Rebuild All button.

Also, when you update the build tools to a new version, you will need to “save” the project and then rebuild all.

3 Debugging

Portos has its own features that help you debug your project. This section is specific to CCS. If you are not using CCS then you cannot currently use the features presented here. We will provide debugging functions for other tools soon.

When Portos catches an error, it is by default configured to halt the program and display an error number. To view the displayed error number, open the BIOS' Message Log window by clicking on **DSP/BIOS → Message Log** (select Log Name: Execution Graph Details, i.e. where the system log is displayed). The error numbers and their description are listed in the document *Portos User and Reference Guide*.

In the default configuration, when Portos catches an error, it prints the error number and then aborts the program. Currently, Portos does not print the file name and line number where the error may have originated. This feature maybe added to Portos in the future. But the best way to debug errors is to place a breakpoint where Portos catches the error. In the header file **`po_error.h`**, place a breakpoint inside the function **`po_abort`**. When Portos catches an error the program stops at this breakpoint. By checking the stack content, you find the list of functions where the error may have originated. You can check the stack content by opening the window **View → Call Stack**.

3.1 GEL Commands

In addition, we have created a set of GEL commands that allow you to access, from the CCS environment, some useful data in your program as you debug it. In order to use the GEL commands, you need to load the GEL file

```
<portos_path>\portos\target\tibios\portos.gel
```

Load this file via the menu **File → Load Gel**. You may need to load this file every time you restart CCS, unless you configure CCS to automatically load **`portos.gel`** at startup (cf. CCS help on "Auto-executing GEL Functions at Startup").

You also need to enable the GEL Toolbar for fast access to GEL commands. To do so, click on **View → GEL Toolbar**.

Now, after you stop your program at some breakpoint, you can use one of the GEL commands to display information:

```
po_memory(int region)
po_priority()
po_function()
po_signal(int group)
po_time(int clock)
po_queue(po_queue_Queue *queue)
```

The command names are self explanatory. You can try them to see what they display.

For example, if you type in the GEL Toolbar the command **`po_memory(0)`**, then the content of dynamic memory region 0 is displayed in a GEL window. In particular, it displays all blocks currently allocated in this region along with source file name and line number where the block allocation was made.

Sometimes the access to the DSP memory is slow and you may need to be patient for the GEL command to terminate uploading all the info if too many blocks are allocated. It prints information as it uploads, and when it is done, it prints a terminating line.

You can also display all running and all waiting priority functions by typing in the GEL Toolbar the command **`po_priority()`** or **`po_function()`**. The two commands are equivalent (they're aliases for the same command).

You can display the priority functions that are scheduled for clock 1, for example, by typing the command **`po_time(1)`**. Or all priority functions attached to signal group 0 by typing **`po_signal(0)`**.

You can display all priority functions waiting in a queue **`MyQueue`**, for example, by typing **`po_queue(&MyQueue)`**. Often, depending on what the compiler does, **`MyQueue`** has to be a global variable to be visible by the GEL command. A static variable may be invisible. In the case of a static variable, you could enter the queue address if you know it, for example, **`po_queue(0x80005678)`**.

Most of the commands don't work in the Release configuration. Either because certain debugging features are turned off in this configuration, or because the GEL interpreter can't seem to work properly when the C compiler optimizes the code.

4 Configuration

The document **`Portos User and Reference Guide`** contains most of the configuration information you need to know about. In addition, there are a few issues specifically related to the TI BIOS case.

4.1 Stack Size

The SWI stack size may need to be increased in size since all your priority functions will be using this stack. You can increase the stack size by opening your **`.tcf`** file. This file is located near your Source files, under the folder DSP/BIOS Config. Open this file. Then open folder System. Then right click on subfolder MEM – Memory Section Manager and open the Properties window. There, select the General tab. In this tab, locate the Stack Size field and increase the stack size as needed.

You can track if the stack is getting full while you debug your program by displaying the window **DSP/BIOS → Kernel/Object View**, then observing the Stack Peak value of the KNL object (Kernel).

4.2 Priority Levels

In the configuration file **`portos/include/cfg/po_cfg_tibios.h`** you can modify the priority levels used by the priority functions. You can only do so if you have the source code (you cannot modify these fields if you only have precompiled libraries). You can tweak the number of software interrupt levels that are reserved for priority functions. The remaining levels can be used by your own SWI.

The default software interrupt levels used by the priority functions are 2 to 12. This is a total of 11 priority levels, and they are remapped to priority function levels 0 to 10. For priority functions you have to use the priority levels 0 to 10 (not 2 to 12). The macro **`po_priority_MAX`** will be, by default, equal to 10.

4.3 Linker Sections

Currently, the Portos library resides in the typical linker sections, such as **`.text`**, **`.bss`**, etc. Grouping Portos functions and data in separate sections is not yet done. Therefore, you should pay attention to where the linker places the Portos library, in internal or external RAM, for example. If it is in slow RAM your program may run slower.

4.4 Demo programs

There are a number of demo programs available under **`portos/demo`**. These programs assume that you have created an MS Windows environment variable called **`%PORTOS%`** and that is set to the path to the Portos directory (e.g. **`PORTOS=C:\proj`**).

5 Precompiled Libraries

Portos provides a set of precompiled libraries. If you don't have the source code, then you must use one of these libraries (if you have the source code, then you can create your own libraries with any build options you choose).

The precompiled libraries can be found under `portos/lib`. The appropriate library should be added to your project (**Build Options → Linker tab → Libraries**).

For the versions of the Texas Instruments *Code Generation Tools* (*Build Tools*) used to generate the libraries, cf. file `portos/lib/po_compilers_versions.txt`. The libraries will be constantly updated as the Code Generation Tools are updated.

Library	Compiler Options	Description
Platform C6x		
`po_tibios6x_lad.lib`	`-g -d"_DEBUG"`	Little endian Aggregate model Debug config
`po_tibios6x_lar.lib`	`-o3`	Little endian Aggregate model Release config
`po_tibios6x_lnd.lib`	`-g -d"_DEBUG"` `--mem_model:data=near`	Little endian Near model Debug config
`po_tibios6x_lnr.lib`	`-o3` `--mem_model:data=near`	Little endian Near model Release config
`po_tibios6x_bad.lib`	`-g -d"_DEBUG" -me`	Big Endian Aggregate model Debug config
`po_tibios6x_bar.lib`	`-o3 -me`	Big Endian Aggregate model Release config
`po_tibios6x_bnd.lib`	`-g -d"_DEBUG" -me` `--mem_model:data=near`	Big endian Near model Debug config
`po_tibios6x_bnr.lib`	`-o3 -me` `--mem_model:data=near`	Big endian Near model Release config

Platform C55x		
po_tibios55x_ld.lib	**-g -d"_DEBUG" -ml**	Large model Debug config
po_tibios55x_lr.lib	**-o3 -ml**	Large model Release config
Platform C28x		
po_tibios28x_ld.lib	**-g -d"_DEBUG" -ml -v28**	Large model Debug config
po_tibios28x_lr.lib	**-o3 -ml -v28**	Large model Release config