olsonet Communications

Pawel Gburzynski

PIP:

an integrated SDK for PicOS

version 1.5

October 2021

Other related documents available from Olsonet Communications:

[picos]	Programming under PicOS
[vuee]	VUE^2: the Virtual Underlay Execution Engine
[side]	SIDE/SMURPH: a Modeling Environment for Reactive Telecommunication Systems, Reference Manual
[installation]	Installation and Quickstart
[picomp]	PiComp: the PicOS Compiler
[vneti]	VNETI: Versatile NETwork Interface
[mkmk]	mkmk: the PicOS Makefile Maker
[mspdebug]	MSPDebug man page
[serial]	UART Communication via VNETI (TCV)
[emspcc11]	EMSPCC11 brochure
[mspgcc]	mspgcc: A port of the GNU tools to the Texas Instruments MSP430 microcontrollers

Preamble

PIP is an integrated SDK for PicOS programmed in Tcl/Tk. This document does not describe everything about PIP. Being a GUI program, PIP needs no detailed explanation of every single item in every menu. Thus, it is merely an introduction. Some selected items that may be difficult to grasp at first sight, are covered in more depth.

1 PIP under Windows (Cygwin)

You may skip this section if you are only going to use PIP under Linux. While Linux would be (and should be) the preferred environment for PIP (and for the rest of our platform), the reality is that we still use Windows + Cygwin quite a bit. Within that framework, PIP is necessarily a kludge operating in the twilight zone between UNIX and Windows. Well, Cygwin is a kludge to begin with, and PIP merely pushes the idea to a new dimension.

Probably the most painful problem of Cygwin is the need to convert file paths between UNIX and Windows format in all those situations when they must be sent to external programs (or are received from them). Before Tcl 8.5 became available under Cygwin (which happened around Cygwin version 1.7.10) PIP (with all its windows, dialogs, and menus) operated under an external (Windows-native) version of Tcl/Tk (courtesy ActiveState)[1] in the Windows context, i.e., formally it needed no Cygwin to live. That created some confusion for at least two reasons:

[1] www.activestate.com.

1. File paths used by PIP were in Windows format, while most (but not all) of the tools invoked by PIP used the UNIX format.
2. Some programs internally invoked by PIP, notably its text editor, needed X11, which ran under Cygwin.

Some of this confusion is still there, especially that ActiveState Tcl/Tk is still a (discouraged) option. On the other hand, many of those early problems have been eliminated or patched out in the present version, so all the different and not always perfectly compatible components almost feel like a harmonious union of homogeneous tools.

1.1 X Windows issues

Even though there exist several alternative X-Windows systems for Windows, PIP needs Cygwin (its set of utilities) for many other things as well. Consequently, using the X11 server that comes with Cygwin always looked like the most natural and obvious choice. Because the windows created by PIP belonged to two different frameworks (X11 + Windows-native), it all only made sense when the two frameworks could coexist dynamically on the same screen. That basically forced one to run the X11 server in the *multiwindow* mode, i.e., without the background window covering the entire screen (known as the *windowed* mode). This is because, in the windowed mode, whenever you want to see a window belonging to the X11 system, you must explicitly hide all native windows (resuming the X11 pane), and vice versa, which is extremely annoying.

When PIP runs under Cygwin's version of Tcl/Tk (which is the default and recommended option), those problems are largely avoided: PIP's windows belong to the same X11 system as most of the remaining windows displayed by the platform. Thus, you can comfortably run most things under X11, also in windowed mode. However, occasionally, PIP may want to invoke some external GUI program executing solely under Windows (like a flash loader from TI), so it seems that the multiwindow mode of X is still more convenient.

Assuming a full and reasonably up-to-date Cygwin installation, the recommended way to start the X11 server in the multiwindow mode is to execute xlaunch (to be found in C:/cygwin/bin).[2] You can create a desktop shortcut to that program. It makes sense to generate and save the configuration file on the first run (otherwise xlaunch will keep nagging you with the initial menu on every startup), store that file in some convenient location and edit the link to xlaunch (right-click→Properties) to make the “Start in” field look like: C:\cygwin\bin\xlaunch.exe -run *config* where *config* is the full path to the saved configuration file. Note that the path will be interpreted by Cygwin (relative to the Cygwin root).

Having started, the X11 server reads the configuration file, .XWinrc, in your (Cygwin) home directory. If no such file is present, it behaves in some default way which is not bad. It shows an icon in the Windows task bar, which you can right-click to open xterm windows (and possibly other stuff), as well as to terminate the server. There is no need to access Cygwin utilities any other way, so you can forget about the (non-X11) Cygwin terminal.

Here is my personal and trivial version of .XWinrc (which is not far from the default one):

```
# XWin Server Resource File (simplified by PG)
menu root {
       "Reload .XWinrc"  RELOAD
       Separator
       xterm             exec "xterm -ls"
```

[2] Or wherever Cygwin has been installed.

```
        Separator
}
RootMenu root
SilentExit
DEBUG "Done parsing the configuration file..."
```

As you can see, the xterm client is practically the only application accessible from the menu (it is invoked with -ls to treat the shell running in the terminal as the login shell). I have removed the indirection required to access xterm under the default setup which you can find (for useful comments and reference) at /etc/X11/system.XWinrc.

1.2 The Windows-native Tcl/Tck option

Even though there is no need these days to go beyond Cygwin to run PIP (and friends) under Windows, the option to use the external (non-Cygwin) Tcl/Tk environment is still there. When the package is deployed [installation] this way:

```
deploy -l
```

then all the GUI scripts of the platform will opt to use the ActiveState version of Tcl/Tk instead of the version that comes with Cygwin. This means that those windows will show outside the X11 server of Cygwin as being Windows-native. Of course, the ActiveState Tcl/Tk package must be installed for that.

Note that, even in the Windows-native mode, Cygwin's X11 server is still needed to display the windows of some programs invoked by PIP, notably the elvis text editor. This means that even though formally you can use PIP without X11 in the native mode, the platform will not be fully functional without X11. The Windows-native mode is discouraged these days. Many things (notably pre-builds, Section 3.2) work there much slower than in the Cygwin-native mode because of the incessant need to convert file paths between the DOS and UNIX formats. Another annoying problem surfacing in the Windows-native mode (which I couldn't do much about) is that the text editor's (elvis) windows (and other windows created under X11 by PIP's requests) cannot reliably negotiate their way up through Windows-native windows, so they may open (partially or completely) obscured. This is probably because X11 windows are considered second-class citizens by the Windows system.

2 Invoking PIP

Just type:

```
pip
```

in a terminal. Alternatively, you can enter:

```
pip -D
```

(capital D), if you want to see the debug messages produced by PIP on its standard output which may be useful for spotting problems.

There is nothing wrong about multiple instances of PIP running at the same time for as long as they don't operate on the same project. There are no safeguards (locks) to prevent that but messing things up this way requires a devious intention.

Generally, PIP tries to be permissive, quite likely beyond the safety zone. During some actions that take more time than just a click (like a compilation, for example), most of the action buttons will allow you to do other things in parallel with the operation in progress. For as long as there is no interference, things will remain sane; however, it is not impossible to mess them up. PIP

does try to assess the sanity of data, often redundantly, to prevent crashes, but even if it doesn't fail formally, things may get messed up. For example, changing the board configuration halfway through a compilation is probably not a good idea.

3 A walk through a simple project

We assume that everything has been set up already. A separate document [installation] explains in detail what must be installed. Go there first if you are setting things up all by yourself.

In this section, we shall create a simple project from scratch and go through all the essential steps of its completion. For the praxis,[3] we shall use the "Hello World" program from Section 2.1 of [picos]. Even though this praxis already has an implementation (to be found in Apps/EXAMPLES/INTRO/HELLO_WORLD), we shall create it from scratch. We will end up with something slightly more complex than the existing version.

3.1 Creating a new project

Start PIP, click File→New project and choose a directory. This will be a new directory, which you can put (basically) anywhere within the filesystem hierarchy. The dialog that opens after the click is set by default at directory PICOS/Apps, which is the standard default container for project directories. Create a new directory there and name it, e.g., HELLO_WORLD.

> **Note 3.1.1:** if you are on Linux or on Windows/Cygwin running PIP under the Cygwin version of Tcl/Tk, the dialog shows no button to create a new folder (directory). You create a new directory by typing its name at the end of the path shown at the bottom of the dialog and hitting Return. If you are under ActiveState Tcl/Tk (in the Windows native mode), you will see a standard Windows file browser dialog.
>
> **Note 3.1.2:** project file/directory names cannot include spaces (PIP won't let you use them, anyway, but it cannot prevent you from creating ones spuriously under the standard open file dialog of Windows).[4] They are considered illegal by PIP, because they confuse some of the tools. This may be fixed at some point, but it isn't a high priority item. Note that under Windows this may be occasionally restrictive, but when we assume that a project directory belongs to the Cygwin hierarchy (for example, it is a subdirectory of PICOS/Apps), then we are practically always safe, because Cygwin paths typically look like C:\cygwin\home\username\..., and it is not natural for UNIX file paths to contain spaces (unlike some Windows systems).

When you select (or type-in) the directory name and close the dialog, a new (project type) dialog will pop up: you must decide whether the project is for a single program or for multiple programs. A multiple-program project is for those cases where the praxis needs nodes of different types, i.e., running different programs, which nonetheless belong to the same application (and should be treated as members of the same project). In a situation like that, you must assign labels (alphanumeric keywords) identifying the different programs (see Section 10.1). As our simple project is going to comprise a single program, just press the Single program button and the dialog will go away.

[3] This is our term for a PicOS application.

[4] The way it works under Windows/Cygwin with native Tcl/Tk is that you first create a "New folder" (the name obviously contains a space in the middle) which you are then supposed to rename to something meaningful. So, the dialog gives you the wrong hint.

The title of PIP's root window will now be showing something like “PIP 1.5, project HELLO_WORLD”. The left pane will present the initial list of the (relevant) files in the project's folder. For a new project, PIP puts there two files acting as stubs (to be filled out), but those stubs already constitute a compilable, loadable, and executable project (which does nothing, however). The two files are app.cc (the root source file) and options.sys (local re-definitions, if any, of the various parameters related to the system and to the node's hardware). This is the formally minimal set of files for any (single program) praxis. The second file, options.sys, is in fact optional; we will keep it empty, anyway, which is equivalent to its absence.

The files in the left pane of PIP's main window (we will be calling it the *tree view*) are organized into five groups named Headers (.h files), Sources (.cc files), Options (options files), XMLData (VUEE data files), and Scripts. Later you may learn that there can be more than one options.sys file (for a multi-program praxis). The data files (their names typically ending with .xml) pertain to the VUEE model of the praxis.

In addition to the above groups, which represent the project's files, i.e., ones stored in the project directory that you have just created (or its subdirectories, if any), the tree view may also show the “board” directory including various files describing the device for which the project has been configured (as explained below). When the project consists of multiple programs destined for several different boards, the directories of all those boards will show up at the bottom of the tree view.

3.2 Compiling for real hardware

The project can be compiled right away and even loaded into a device, although the latter action is pointless, because the program does nothing. Nevertheless, let us try to compile it to learn the drill. When you click on the Build menu, you see that the only options available are for VUEE, i.e., for the emulator [vuee]. This is because you haven't assigned a board (device) to the project yet, so PIP doesn't know how to compile it for real hardware. To do that, click Configuration→Arch+Board which will produce the Architecture/Board selection dialog. If you are compiling for EMSPC11, select MSP430 as the architecture and WARSAW for the board (you must do it from a two-level menu: WARSAW …→WARSAW). For the CC1350 Launchpad, select CC13XX and CC1350_LAUNCHXL, respectively.

Click Done. The tree view will now show (underneath the previous set) a new entry (with a distinctive gray background) looking like the name of the selected board. When you open it, you will see the list of standard headers describing the device of your project. If you look at the Build menu again, you will see that it now offers two more choices appearing at the very top. Assuming you have selected the WARSAW board for MSP430, they are: Pre build (WARSAW src) and Build (make).

The detailed procedure of creating the loadable image of a program is described in [mkmk]. For now, suffice it to say that the operation is carried out in two steps: pre-building, which corresponds to configuring the program for the specific environment, and the actual compilation. The outcome of the first step is a Makefile[5] to be used in the second step.

So, when you click Pre build, you configure the project for compilation. You will see in the right (console) pane of PIP's main window the messages produced by mkmk, which has been invoked to do the job. At the same time, the stripe above the tree view pane (the status line), which normally says “Idle,” will become “Running,” and an advancing seconds counter will

[5] For a multiple-program praxis, there may be more than one Makefile.

show up to the right. This means that some program is now running and (possibly) writing its output to the console. When mkmk finishes, the text “--DONE—" will appear at the end of the console's output and the status line will say “Idle” again.

> **Note 3.2.1:** a program running in the console usually does not block PIP, in the sense that those features that can be sensibly used in parallel with that program are available (see Section 2).
>
> **Note 3.2.2:** a program running in the console can be aborted by selecting the Abort entry from the Build menu (the very same action is also available from the Execute menu). The entry becomes enabled whenever the status line says “Running.”

The standard (and most flexible) way of compiling a PicOS praxis, the *source mode*, involves the compilation of all system (PicOS) files needed by its program(s), including the kernel source code (see [mkmk] for details). This is because those files are heavily parameterized by various configuration constants whose purpose is to make sure that only the code fragments that are in fact needed are compiled in. This approach is OK, because the system is small enough to compile fast. Of course, once compiled, the respective files will only be recompiled when their dependencies change, or when you “clean” the project, reconfigure it, or pre-build it, which action also removes all previously compiled object files.

An alternative way to compile a praxis, the *library mode*, involves an object library prepared in advance and associated with a specific board. The choice between the *source* and *library* modes is applicable to individual programs of the praxis; in a multiprogram case, it may deal with multiple libraries prepared for the different boards configured with the different programs. The selection is determined by the Lib mode checkbox in the Board selection dialog (mentioned at the beginning of this section). We need not concern ourselves with this issue right away.

Note that the Build menu shows two cleaning buttons: Clear (full) and Clear (soft).[6] The first button removes all files that can be trivially recreated, including the Makefile(s) procured by pre-building. The second button works more like a traditional case of “make clean”, i.e., it removes the object files, but doesn't touch the Makefile(s), so the soft cleanup doesn't destroy the outcome of a pre-build.

For our present exercise, the next step is to compile the praxis. When you click Build (make) in the Build menu, you will effectively execute make on the Makefile created in the previous step. The console will show the messages produced by the compiler and the linker. As before, the completion is signaled by the text “--DONE—" appearing at the bottom of the console output; at the same time the status line reverts to “Idle.”

We did the two steps explicitly for education; however, you would have accomplished the same feat by clicking Build (make) directly. If no Makefile is present when a build is requested, PIP will carry out the pre-build action automatically. Note, however, that a manual pre-build is always effective, in the sense that the full action of constructing the Makefile from scratch is carried out regardless of the status of any existing Makefile. Also, pre-building removes the entire output of a previous build.

[6] The selection for a multiprogram praxis is wider as the soft cleaning can be carried out on a per-program basis.

3.3 Editing files

Technically, the binary program produced in the previous step can be uploaded into the device. It doesn't make a lot of sense, however, because the program exhibits no activity that you could see. Thus, we shall beef it up a bit before proceeding with that step.

Double click on the app.cc file (under Sources) in the tree view pane. This will open the file for editing in a separate window. Editing in PIP is handled by elvis which is a moderately popular VI-compatible, X11-capable, and quite powerful text editor written (some longish time ago) by Steve Kirkendall and adapted by me to collaborate with PIP. It comes with extensive (built-in) documentation and is highly configurable. Most of the interesting configuration options of elvis are available from PIP's Configuration menu.

The stub in app.cc consists of a trivial root FSM [picos] with a single empty state. Edit it out to look like this (feel free to be more creative):

```
#include "sysio.h"

fsm root {

	int led = 2;

	state INIT:
	    ser_out (INIT, "Hello world\r\n");

	state ROTATE_LEDS:
	    // Previous led goes off
	    leds (led, 0);
	    // Increment the led number modulo 3
	    if (++led > 2)
		led = 0;
	    // The led goes on
	    leds (led, 1);

	    // Wait for one second
	    delay (1024, ROTATE_LEDS);
}
```

If you pre-built the program in the previous step, there is no need to do it now, because, despite all that editing, you haven't changed anything that would affect the structure or configuration of system files needed by the program. For as long as you:

1. create no new files within the project
2. insert no new #include headers into the existing files
3. do not change the values of symbolic constants in options.sys
4. do not reconfigure the project (e.g., change the board)

there is no need to rebuild. In principle, the above situations could be detected by PIP/elvis and used to trigger pre-builds automatically, but I prefer to leave the decision to your discretion, the underlying philosophy being that giving power to experts makes more sense than trying to

comfort all clueless people.[7] There is little penalty for pre-building “just in case”, so in all those situations where you are not sure, just hit the button.

Regardless of what you do next, i.e., pre-build or build (compile), if you haven't saved the file before selecting the action in the menu, PIP will warn you that a file has been modified but not saved and ask what to do. Your options are to save the file before continuing, continue with the old version of the file, or abort the operation. Of course, you can save the file explicitly from the editor window as well. Another thing that you may have noticed is that the tree-view name of a file being edited changes color to green or red (the latter means that the file has pending, unsaved modifications).

Close the editor window and try to build the program (press the second button on the Build menu). You will get an error message from the linker saying this:

```
/home/pawel/SOFTWARE/PICOS/Apps/HELLO_WORLD/app.cc:8: undefined
reference to `ser_out'
```

Double click on the error line in the console pane. The editor window (for app.cc) will pop up with the cursor pointing to the offending line. The problem is caused by the lack of a pertinent header file that would indicate to PIP (specifically to mkmk) that the library file containing ser_out should be compiled in. We shall address this problem in a little while.

3.4 C-tagging

Our program is extremely simple: it basically consists of a single file whose entire contents can be seen at once in a single (not so big) editor window. Nonetheless, we can use it to illustrate the search capabilities of PIP and elvis.

First let us complicate the program a bit by adding a second file (to have the simplest possible case of many). We shall encapsulate the code for rotating LEDs into a function. For that, let us create a new file named leds.cc. Right click anywhere within the Sources section of the tree view pane (e.g., on app.cc). In the menu that pops up, hit New file and name the file leds.cc. An editor window will open. Make the file contents look like this:

```
#include "sysio.h"

void rotate_leds () {

        static int led = 2;

        leds (led, 0);
        if (++led > 2)
            led = 0;
        leds (led, 1);
}
```

Then replace the corresponding piece in app.cc with a function call:

```
#include "sysio.h"

fsm root {

        state INIT:
            ser_out (INIT, "Hello world\r\n");
```

[7] One exception is reconfiguring the project, which action forces Clear (full), thus automatically forcing a pre-build.

```
        state ROTATE_LEDS:
            rotate_leds ();
            delay (1024, ROTATE_LEDS);
}
```

Now save the files and close both editor windows, pre-build, and then build.[8] Now you will get two errors (note that we haven't fixed the previous one yet, so it is bound to pop up again):

```
/home/pawel/SOFTWARE/PICOS/Apps/HELLO_WORLD/app.cc:9: undefined
reference to `rotate_leds'
/home/pawel/SOFTWARE/PICOS/Apps/HELLO_WORLD/app.cc:6: undefined
reference to `ser_out'
```

When you double click on the first of the two error lines, the editor window for app.cc will pop up pointing to the rotate_leds call. To fix this problem, you must make sure that the second source file, the one you have just created, will also compile as part of the project. PicOS (or mkmk, to be exact), requires that those project files that should take part in pre-building and later compile into the project's uploadable image be explicitly referenced from other files that are already marked for compilation (see Section 3.4 in [mkmk]). The only file compiled by default is app.cc. One way to make sure that leds.cc is compiled as well is to request it from app.cc by inserting (anywhere in the file) a comment looking like this:

```
//+++ leds.cc
```

Note that there must be no space between // and +++. In most cases, such special comments are put into header files and those header files are then included by the source files already belonging to the project, with app.cc acting as the root of the tree. In our simple case, we will just add the above line to app.cc right after the first (include) statement and additionally insert there a header of rotate_leds, i.e.,

```
//+++ leds.cc
void rotate_leds ();
```

such that the external function is properly defined. Suppose that you want to have a look at rotate_leds before fixing the problem (and your program is not as trivial as our example). If you double click on rotate_leds in the editor window of app.cc, PIP will open the window for leds.cc with the cursor set on the definition of rotate_leds. This is called *C-tagging*. PIP builds (and re-builds at relevant stages) its database of C-tags of all .cc and .h files belonging to your project (those listed in the tree view pane of PIP's main window) reflecting the locations in those files where things are defined, declared, and so on. Not everything is tagged this way, e.g., local (automatic) variables or functions and FSM states are not.[9] Notably, system (i.e., PicOS) files can be tagged as well. For example, when you double click on one of the calls to leds in rotate_leds, PIP will open a (read-only) editor window with the system file sysio.h pointing you to the respective macro.

3.5 Access to system files

Click Configuration→Options. You will see a dialog defining a few general parameters associated with your project. The two menus (labeled "Show and edit ...") allow you to declare:

[8] Note that this time you have to pre-build, because you have added a new file to the project.
[9] They are typically easy to see in their usage context.

- which PicOS files you want to see (and when) while navigating through the sources of your project
- which VUEE files (i.e., ones belonging to the VUEE simulator) you want to see

Note that VUEE is a separate "system" with its own source files implementing SIDE models [side] of the PicOS stuff and the network environment [vuee]. If you are running and debugging your program virtually under VUEE, you may be interested in seeing the interiors of that model, rather than the interiors of PicOS.

The options regarding your access to system files (defined separately for PicOS and VUEE) are:

> **Never** → the system files are never C-tagged and never searched by PIP (with the search engine described in Section 3.6).
>
> **Tags, R/O** → the system files are C-tagged, but not searched. When referenced by C-tags, those files will open read-only (so you won't be able to modify them inadvertently).
>
> **Always, R/O** → the system files are C-tagged and searched. They always open read-only.
>
> **Always, R/W** → the system files are C-tagged, searched, and always open read-write.

The last option is for PicOS development. Note that the protection offered by PIP's configuration options is mostly advisory. You can always edit a system file from outside PIP and do with it whatever you please. Also, a read-only file can be overwritten by elvis with the w! command, if you have formal write access to the file within the file system.

The default settings are **Tags, R/O** for PicOS and **Never** for VUEE. Note that they apply on a per-project basis, unlike, e.g., color schemes for elvis (Configuration→Editor Schemes …), which are stored globally (for the user).

> **Note 3.5.1:** if you change the setting for PicOS system files to Never, the board directory entry (or the multiple board entries, for a multiprogram project) will disappear from the tree view. Formally, board descriptions belong to the system, so if you declare that you do not want to view the system files at all, PIP will hide the board definition(s) from the view.

The C-tags database for your project files is updated whenever any of those files is modified (or created), but not until the file is saved by elvis. For system files, the situation is a bit more complicated (and different rules apply for PicOS and for VUEE).

For PicOS, only those system files that the project needs are C-tagged. Their set can only be known after the project has been pre-built. Consequently, no PicOS files are accessible via C-tags clicks if the project hasn't been pre-built yet, or after it has been fully cleaned: Build→Clean (full). In the more complicated case of a project consisting of multiple programs, the list of PicOS files to be C-tagged is determined from all the pre-builds combined (by inspecting the current collection of the project's Makefiles).[10]

For VUEE, assuming the C-tags option is on, the C-tags are computed once for all, for the complete set of all VUEE files. These are the files in the VUEE/PICOS directory of the VUEE package.

[10] When the library mode is used, the list of system files is determined based on the library content.

3.6 Searching

We are now ready to try to attend to the first problem, i.e., the missing header file for ser_out. A simple idea to find that file's location might be to double click on the call to ser_out in app.cc. But nothing happens when you do that, except for a line showing up in the console pane to tell you: "Tag ser_out not found".[11] Why not? Because, as explained in the previous section, to be C-tagged, a PicOS file must be referenced from the project, and this is exactly our problem: the file is not referenced!

In a situation like this you may want to resort to searching. Click File→Search to bring up the search window. Select WD and check Case. The first widget determines the interpretation of the search string (as a word, or a sequence of words), the second says that the case of letters matters. You may select Only from the Sys list to indicate that you want to search *only* system files. Then type ser_out into the String field. Alternatively, you can double click once more on the function name in app.cc. When the search window is open and the tag is not found, it is automatically inserted as the search string. Now press the Search button.

You will see a lot of matches looking mostly completely uninteresting (lots of params.sys files in board definitions). We can easily narrow down the search to what we really care about, i.e., the header files. For that, enter \.h$ into the FN Pat field. This is a regular expression to be matched against the file name. Clear the output area with the Clean button and try again. This time it looks much better. There is less than a handful of matches, and you can clearly see that the requisite header file is:

```
…/PICOS/PicOS/PLibs/Serial/ser.h
```

thus, you must insert:

```
#include "ser.h"
```

somewhere at the beginning of app.cc.

Every match presented in the output pane of the search window begins with a highlighted header listing the file name and the line number in the same format as the error locations produced by the compiler. When you double-click on such a header, and the current option settings (Section 3.5) allow you to view the file, an editor window will pop up with the file positioned at the respective line. Every match is shown in the search window as a range of lines whose span is determined by the Bracket field from the lower line of the collection of widgets at the bottom. The default setting of 5 means that the range will include 5 lines in addition to the matching one (i.e., 6 lines altogether), with the matching line located approximately in the middle. The matched string is highlighted within the line.

With word matching (selected by WD) it doesn't matter how many spaces appear between the function name and the parenthesis; what does matter instead is the exact succession of *words* with any blanks between them ignored. The three options for the interpretation of the search string are:

RE → regular expression[12]

[11] If you have changed the viewing option for VUEE files to Tags, R/O or Always, you will get in this step the VUEE definition of ser_out, which is not what you want.

[12] See, e.g., http://www.tcl.tk/man/tcl8.6/TclCmd/re_syntax.htm for a detailed description of regular expressions in Tcl.

ST → string, i.e., simple string matching without interpreting any characters as special

WD → a sequence of words, possibly separated by blanks (but no newlines)

For the last option, a word is either a sequence of consecutive letters and/or digits and/or the underscore character, or any other character standing by itself for a complete word. Thus, for example:

```
words @#$ 11@( _ABC1/
```

contains 9 words:

```
words, @, #, $, 11, @, (, _ABC1, /
```

which will match any of these:

```
words   @#   $11@(_ABC1/
words@  # $ 11 @  (  _ABC1 /
```

but not:

```
words@#a$11@(_ABC1/
words @ # $ 11 @ ( _ABCD1 /
```

Whether the case of letters is relevant or not depends on the Case option.

The Sys selection allows you to choose whether you want to search through the PicOS system files, with the VUEE checkbox optionally throwing in the VUEE files to the system pool. The options are:

None → only the project files will be searched (no system files at all)

Proj → only the system files associated with the project (based on the current pre-build) + all VUEE files, if the VUEE box has been checked, will be searched

All → all system files (pertinent to the project's architecture) will be searched

Only → *only* all system files will be searched, i.e., not the project files

Note that in the second case the project must have been pre-built (see Section 3.2) for the PicOS files to be searchable (only then can PIP know the collection of files to search).[13] This does not apply to VUEE files (if the VUEE box is checked), which are always searched *all*.[14]

The lower widget line of the search window contains action buttons and fields. The role of those fields is to limit things. For example, the already described Bracket field limits the width of the displayed neighborhood of a matched line. Max lines determines the maximum number of lines stored in the output area (defaulting to 1000). Max cases limits the number of matches (the default limit is 256); the search stops when that number is reached.

By putting a (Tcl) regexp into the FN Pat field you restrict the population of files to those whose names match the specified pattern. This field is typically used to limit the search to source or header files. By checking the ! box in front of the field, you negate the interpretation of the pattern, i.e., it will identify the files that *will not* be searched.

[13] For a program built in the library mode, no pre-build is necessary, because the set of PicOS files needed by the program is determined based on the library content.
[14] This reflects the fact that all those files are compiled into every VUEE model.

The colors used to highlight the headers of the matched sequences presented in the output pane, as well as the matching fragments within those sequences, are configurable with the H and M buttons. Just press them and see what happens.

The meaning of the action buttons Search, Clean, and Close is rather obvious. The remaining two buttons, Edit and New, allow you to edit and create any file from PIP, not only a file related to the project, which may be useful when developing PicOS or VUEE. With XTerm you can open an X11-terminal within a selected directory. With Explorer, you will open the GUI file explorer instead.

3.7 Uploading programs into devices

Let us fix our program, i.e., make sure that these lines have been added to app.cc:

```
#include "ser.h"
//+++ leds.cc
void rotate_leds ();
```

Now we have to pre-build because we have added a new header file to the project. When we do that and subsequently build the program, there will be no more errors. The program is ready to be loaded into the device.

For the first exercise, we assume that the project has been compiled for the MSP430 WARSAW board (e.g., EMSPC11) whose connection to the PC consists of the FTDI TTL232R USB UART dongle and the MSP-FET430UIF programmer (via USB cables).

Connect the two devices and make sure that the board is powered up. Click Execute→Start piter to open a terminal window that will allow you to communicate with the board via its UART [serial] once the device gets to running the program. Click Connect in the piter window and select the respective serial (COM) device of the FTDI dongle. If you don't know it, hit the Scan button; then the device menu will only show those serial devices that can be opened. Note that with PIP running under the Cygwin version of Tcl (and of course under Linux), the devices are displayed in a UNIX-like manner, e.g., COM3 becomes /dev/ttyS2 (the tty number is equal to the COM port number minus one). On Linux, the device may be named like /dev/ttyUSB0 or /dev/ttyACM1 where the value at the end may differ. The dongle is usually easy to spot as something new and different from what you are used to. On Cygwin (under Windows), the same dongle will tend to map to the same COM device on subsequent connections. This isn't the case on Linux where the device number reflects the dynamic connection order. Select the speed of 9600 bps and the Direct protocol, also make sure that the Bin box is not checked (these should be the defaults, anyway). When you hit Save & Proceed, your settings will be saved inside the project, so the next time around you will see them as the new initial values of piter's parameters.[15]

Click Configuration→Loaders to select the loader program. The way to configure loaders is explained in Section 4; for now, the initial (default) configuration should include MSPDebug [mspdebug]. The dialog should show you "Loader 0" (MSPDebug set to act as a flash loader) and "Debugger 1" (the same MSPDebug configured as a proxy for GDB). Make sure that the radio button labeled "Use" for "Loader 0" is checked, i.e., the loader guise of MSPDebug is selected.

[15] The advantage of this on Linux is dubious because USB device names are assigned in the connection order and are thus likely to be different on different occasions.

Click Execute→Upload image … in the PIP menu. You will see in the console the messages produced by MSPDebug. It will likely want to update the firmware in the programmer (MSP-FET430UIF) first. If so, this will only happen the first time around (as described in [installation] where a similar exercise is discussed). Eventually, it should write the image into the device and start it. This will produce the line “Hello world” in the piter window, and the three LEDs on the device will begin to blink in turns.

For the second exercise, we shall compile the project for the CC1350 Launchpad. Click Configuration→Arch+Board and select CC13XX (for the architecture) and CC1350_LAUNCHXL (for the board). PIP will clear the project (erasing the previous build as well as the pre-build). Assuming you have installed the requisite ARM toolchain [installation], you can now build the image for the Launchpad in the same way you did it for EMSPC11.

One little thing you may want to change in the program is to account for the fact that the Launchpad only has two LEDs, but it isn't necessary. The third (nonexistent) LED will simply never blink.

The most potent flash loader for CC1350 is UNIFLASH by Texas Instruments [installation]. One of its disadvantages is that it is a monstrosity; the other one is that it is fully GUI, so it is impossible to pass it arguments from PIP. You can invoke UNIFLASH from outside of PIP and just follow the GUI. UNIFLASH is formally configured into PIP as a default loader for the CC13xx architecture, although its integration is illusory because all that PIP can do is to run the program on a click (not a lot more than just clicking on the UNIFLASH icon, as if PIP did not exist).

Connect the board to the PC via a USB cable. This time there is a single USB device setting up two serial devices on the PC, one providing the interface to the XDS100 programmer on the board, the other mapping to the device's UART. Start piter as above. You must guess which of the two devices is the UART, but there are only two options, so you will err at most once. Invoke UNIFLASH, either manually (from outside PIP), or by clicking Execute→Upload image with “nw” selected as the loader.[16] The program is sluggish, so be patient. It should auto-detect the Launchpad after a while. Navigate to the project’s directory and point the program to the file named Image.out (this is the ELF version of the uploadable image). If you have goofed with the selection of the UART device, you will see nothing in the piter window, so you should point piter to the other device and try again.

3.8 Building and running a VUEE model

Building a VUEE model of our simple application is quite easy and involves no additional programming. One extra thing we must provide is a data file to the model, which describes such parameters as the population of nodes, their hardware configuration, their spatial distribution, the properties of the radio channel, and a few more items. There is no network in our case (we haven't programmed any radio activity into the node), so we can get away by borrowing the trivial data set from the old HELLO_WORLD praxis that comes with the package.

In the tree view pane, select (left click) the XMLData header and then right click on it. From the menu that pops up select Copy from. Alternatively, having selected XMLData, you can click File→Copy to the same effect.

[16] This happens to be the name of UNIFLASH executable to be found in subdirectory node-webkit of the program’s installation directory.

This will produce a dialog. Navigate to EXAMPLES/INTRO/HELLO_WORLD/data.xml and open that file. It will be copied to your project and appear in the XMLData section as data.xml. When you look at it, you will see that it describes a trivial “network” consisting of one node. Now we are ready to run the program in VUEE.

Click Configuration→VUEE to configure the VUEE model. Check these two items: Do not propagate board config to VUEE and Run with udaemon and select data.xml for Praxis data file. You may also check Terminate when udaemon quits. There is no need to touch the remaining widgets. Click Done.

> **Note 3.8.1:** the reason why you check Do not propagate board config to VUEE is that otherwise PIP would try to base the node configuration of the model on a description associated with the current BOARD selection, i.e., WARSAW. The WARSAW device is equipped with radio, so VUEE would complain about the absence of a radio channel in the trivial network of our project.

Click Build→VUEE to compile the model. When you see “--DONE—" in the console pane (and “Idle” in the status line), the model is ready. To run it, click Execute→Run VUEE.

In udaemon's window that will pop up shortly, type 0 into the Node number field (there is only one node numbered 0 in our network) and click Connect to connect to the node's UART (which is selected by default in the menu to the right). You will see the “Hello world” message in udaemon’s console, as you did before in piter’s window, except that now the message is coming from a virtual device. To see the LEDs, select LEDS from the menu in udaemon's window, the one that was initially showing UART (ascii), and click Connect again, this time to connect to the node's LEDS module.

If you have checked Terminate when udaemon quits in the VUEE configuration window, the model will terminate when you exit udaemon. Otherwise, the model will continue running and you can connect to it again (by starting another instance of udaemon from Execute→Run udaemon), as many times as you please.[17]

Note that VUEE models can be disabled for a project by checking a box in the Configuration→VUEE dialog. This has the effect of dimming/disabling the VUEE-related items in the Build and Execute menus and makes sense for those projects that are not meant to be VUEE-compatible, e.g., hardware tests.

4 Configuring flash loaders

Any flash loader than can be started from a command line can be integrated with PIP, at least to some extent. Perfect integration is only possible if the loader can accept a command line argument telling it which file to upload. But even UNIFLASH, which does not seem to be willing to accept such an argument, can be (somewhat) integrated in the sense that it can be at least started from PIP. You may have noticed that UNIFLASH remembers the location of the last loaded file (in fact a few last loaded files), so its lack of perfect integration is not that painful.

Click Configuration→Loaders … while a project is opened to see the set of loaders defined for the project (for the currently selected architecture). Note that loaders are set up on a per-project, per-architecture basis. When a new project is started, and its architecture is

[17] It is all much more impressive with a network consisting of many nodes, plus the geometry and window preset data file for udaemon, causing the model's complete visualization to pop up at the click of a single button. Read [vuee] for details.

determined, PIP uses a default configuration of loaders for the architecture which is described in file compile.xml in the architecture directory of PICOS (see [mkmk], Section 3.2, for details and Section 8 below).

For illustration, have a look at the set of loaders for our simple project. You can see that there are two types of entities defined in the dialog: loaders (proper) and debuggers. A loader provides a way to write (upload) an image into the device, while a debugger makes it possible to interact with the program executed in the device, which usually also includes overwriting that program with another image. Thus, a debugger may also act like a loader, although a straightforward loader usually does its job in a simpler way and with less hassle for the user.

To debug a program running in a microcontroller, we need a way to convey information between the microcontroller and the actual debugger which is mostly GDB. The interface is implemented by the so-called proxy whose role is to follow the protocol expected by the device and present to GDB something it can understand. Note that MSPDebug, in addition to being able to upload a program into the device, can act as a GDB proxy in one of its functions [mspdebug].

In PIP's configuration dialog, a simple loader is described by two strings: the path to the loader's executable + the string to be passed to the loader as its call arguments. The path can be specified absolutely (starting with /) or as a program callable from the shell, i.e., accessible through the PATH. The dialog allows you to modify the strings defining a loader, create new loaders (the New button), and delete loaders (the Delete button). The best way to revert to the default configuration of loaders for the architecture is to delete all loaders (and debuggers), then exit and re-enter the project.

The sequence %f occurring anywhere in the loader argument string will be replaced by the name of the image file when the loader in invoked. It is possible to restrict the file type and/or change the file extension to suit the expectations/requirements of the loader, e.g.,

```
%f.a43
```

means that only HEX files can be uploaded, while:

```
%f.a43=hex
```

says that the HEX file (original extension .a43) should be renamed (copied) to a file with the extension .hex before submitting it to the program. Such tricks may be needed by some loaders that insist on specific extensions for the loadable files they recognize.

If a loader accepts no arguments, like for example UNIFLASH, then there is no way to pass it the image file on invocation. The loader can still be started from PIP, but you will have to provide it with the file to upload using its own dialogs.

The definition of a debugger includes these components:

- the path to the proxy's executable
- the arguments to be passed to the proxy
- the path to the debugger
- the arguments to be passed to the debugger
- the communication port for the connection between the debugger and the proxy

The argument string for the proxy may include the sequence %p which will be substituted by the port number. The reason why the port number must be defined in a separate widget, as opposed to being directly inserted into the proxy argument string, is that it must also be

provided to the debugger.[18] This is done via the debugger initialization file which is typically called .gdbinit. The file is created by PIP, based on the template described in compile.xml [mkmk], and put into the project directory before the debugger is started.

The argument string for the debugger usually includes the name of the image file to be used as the reference for the debugged program (and possibly to upload into the device). The rules for including that name are the same as for the argument string of a loader. In a typical setup, GDB needs just that file name and nothing else.

The radio button labeled “Use” in a loader description frame selects the loader for use. Only one loader can be selected at a time. This is the one that will be invoked when you click Execute→Upload image …. Note that a debugger is invoked in the same way as a loader.

The simplest (and quickest) flash programmer for CC1350 under Windows/Cygwin is Flash Programmer 2 by Texas Instruments (http://www.ti.com/tool/flash-programmer); unfortunately, it is not available under Linux. Under Cygwin, you can use the GUI version (which cannot be fully integrated with PIP because there is no way to pass it arguments) or you can configure the command-line program that comes with the loader. The executable file of the command-line loader is named srfprog.exe and can be found in the bin directory of the Flash Programmer 2 installation. This is the argument string:

```
-t lsidx(0) -e pif -p -f %f.hex
```

You can learn about those and other arguments of srfprog.exe by running the program (without arguments) from a DOS command prompt.

5 Parameterizing uploaded images

Quite often different nodes of the same network must receive identical images (the nodes run basically the same program) which however differ in some little detail (so the nodes can tell themselves apart). The standard example of such a detail is the Node Id (interpreted as the node address for networking). PIP provides for a simple and reasonably generic mechanism whereby a single compiled image can be cloned into multiple copies (destined for different node specimens) differing in one or two longword (32-bit) constants implanted into the code and available to the program as static, constant data. This mechanism is invoked by clicking Execute→Customize image in the PIP window which opens the flash image generator dialog. At least one image (previously produced by a build step) must be present within the project directory for this selection to be enabled.

The original (vanilla) image file should include at least one (and possibly two) distinctive 32-bit values to be systematically substituted in the cloned copies. Those values should be unique within the image (it is not a good idea to use 0 or 1, but something less likely to occur by chance). By default, the generator assumes that the first value is 0xBACADEAD and the second value is 0xBACADEAF. The first value is the standard tag used by a PicOS praxis to refer to the 32-bit identifier known as the Host Id (or HID) and declared as:

```
const lword host_id = 0xBACADEAD;
```

within the kernel. This constant is global and accessible to the application program. The second 32-bit value is optional, and its interpretation is left to the application. Most of our standard praxes assume that the lower half of HID is the 16-bit Node Id (node address) and it is interpreted as such by TARP (the networking scheme). The upper part of HID is sometimes

[18] In principle, it is possible to run the debugger on another computer connected to the proxy over the network.

used as the 16-bit Network Id (intended to separate different networks operating on the same RF channel).

The parameterization consists in replacing the upper half of HID with some fixed bit pattern (the same for all image clones to be generated) and modifying the lower half (viewed as a 16-bit unsigned integer number) by setting it to consecutive numbers (different values for different clones) starting from some initial value. Optionally, the second value (0xBACADEAF) can be replaced by some fixed 32-bit pattern (the same for all clones). Thus, the only discriminating element for the clones generated in a single go is the lower half of HID (aka the Node Id).

An application willing to take advantage of the parameterizable “second value” should declare a 32-bit constant with the proper preset, e.g.,

```
const lword second_value = 0xBACADEAF;
```

The topmost widget in the generator dialog window is a menu to select the image file format which can be IHEX (for the .a43/.hex Intel HEX format) or ELF. This determines the variant of the source image to be used for the template as well as the format of the target clones to be generated. Obviously, it should agree with the format expected by the loader that will be subsequently used to upload the cloned images into the devices.

Under the type selection menu are two text entry widgets: one pointing to the input image (the vanilla template) and the other describing the pattern to be applied for naming the files containing the cloned images. To select the input file, you can use the Change button (to the right of the text widget) to open a file selection dialog, or directly enter the file's path into the text area. In the simplest case, there will be just one image, e.g., named Image.a43 (automatically inserted as the default selection), but the choice need not be trivial for a multiprogram praxis.

The naming pattern for the clones looks like a file name. If the name includes the sequence “_nnnn” (underscore followed by four lower case letters n), it will be replaced in each output file name by “xxxxxxxx_ddddd” where “xxxxxxxx” is the full, 8-digit hexadecimal value that goes into HID, and “ddddd” is the 5-digit decimal value (with leading zeros inserted as needed) on the 16-bit lower half of HID that gets incremented for every next clone.

The HID Cookie field includes the original HID value to be searched and replaced in the source image. This field is filled with 0xBACADEAD by default, but it can be changed, if needed. The NetId field specifies the upper portion (the more significant half) of HID to be stored in the clone and defaults to zero. The APP Cookie field specifies the original content of the “second value” and APP Value is its replacement. If APP Value is empty (which is the default), the second value is neither sought in the template image nor replaced.

The field labeled From specifies the starting value for the lower half of host_id and How many tells the number of clones to be generated. The minimum initial value of From is 1 (a Node Id cannot be zero).

If the source image contains no (properly aligned) occurrence of the value specified as the HID Cookie, the generator will complain and do nothing. If the value occurs more than once, the generator will object as well (it wouldn't know which occurrence to replace, and it is only allowed to replace one occurrence per clone). The same applies to APP Cookie if APP Value is not empty.

For illustration, suppose that the source file name is Image.a43 (the format is IHEX), the clone file name pattern is Image_nnnn.a43, HID Cookie and NetId are left at their default values,

APP Value is empty, From is 10, and How many is 4. When you click Generate, the program will produce four image files named:

```
Image_0000000A_00010.a43
Image_0000000B_00011.a43
Image_0000000C_00012.a43
Image_0000000D_00013.a43
```

The values inserted into the HID field in the clones can be directly inferred from the file names.

The Compare button in the generator dialog can be used to compare a clone file (identified in the Output field) with a template (the Input field). Its primary purpose is to verify whether an image has been produced by cloning the template (is identical to the template except for the substituted values) and to show the discerning values of the clone.

6 Debugging

When a debugger is selected as the loader to "Use," then in response to Execute→Upload image … PIP will invoke the proxy in the console pane and open a separate terminal window with GDB connected to the proxy. If the board has been flashed with the image, the program running in it can be started, stopped, stepped, break-pointed, and so on, as per the repertoire of GDB commands which can be looked up in several documents easily available on the network. The set of proxy-specific monitor commands makes it possible to flash the device from the debugger. For example, with MSPDebug, you can upload an image file into an MSP430 board with these commands:

```
monitor erase all
load Image
```

Then, when you enter:

```
continue
```

the program will start and run until a breakpoint. For opencod used as the proxy (for a CC1350 device), the flashing command is:

```
monitor program Image
```

Refer to the documentation of GDB and the respective proxies for more information.

The debugger started by PIP is run in a separate terminal window which (by default) involves invoking xterm. This is not the most popular terminal program these days, and I use it mainly for sentimental reasons. The Configuration→Options … dialog has two entries that can redefine the commands to open a terminal window. The entry labeled Shell command in a terminal window is used to start the GDB window. Its default contents are:

```
xterm -e %f
```

where %f is substituted by the command (GDB invocation) passed to the terminal.

7 Running external programs from PIP

As a crude generic solution for problems that are not directly addressable by PIP's widgets, there is a way to execute any (external) command in the project's directory. The bottom line of the console, normally disabled, provides a means to enter input to such programs. Their output will appear in the standard output area of the console.

One special case of an external program is GDB used to debug a VUEE model, as opposed to debugging a program running in the microcontroller (Section 6). To see how it works, start by compiling the VUEE model with debugging enabled. Select Build→VUEE (debug); this will add the -g flag to the options of the SIDE compiler [side] causing debug information to be appended to the model's executable. After the compilation successfully completes, click Execute→Run VUEE (debug). PIP will start GDB in the console window giving it the SIDE executable file as the argument. The bottom (input) line will activate, and you will be able to enter commands to the debugger. Note that regardless of the VUEE configuration settings (in Configuration→VUEE) udaemon is not automatically started in this case. You can always start it manually (Execute→Run udaemon) if needed.

Enter:

```
run data.xml
```

into the console's input line to start the model.

You can abort any program running in the console with Execute→Abort (or Build→Abort, both choices trigger the same action).

When you select Execute→Run program, you will be shown a simple dialog to enter the command you want to execute in the console window. This is potentially dangerous, so you should know what you are doing. In particular, the program can be sh. In such a case, however, a better idea might be to click Execute→Terminal and simply open an xterm client in the project's directory. The terminal will be completely detached from PIP; in particular, it won't go away when PIP quits.

8 Modifying the parameters of toolchain programs

Sometimes it may be useful to change a parameter of the compiler or (temporarily) switch to an alternative toolchain. For example, for low-level debugging, it may be useful to compile the project with optimization switched off, so the machine code will be better matched to the source. With two separate MSP430 toolchains installed side by side, one may want to switch between them without too much hassle.

The way to call the gcc compiler, and other programs of the toolchain, is described in the toolchain configuration file. The default name of the file is compile.xml and it is sought first in the project's directory (where it is usually absent) and then in the architecture directory of PicOS.[19] Note that different toolchain configuration files should be used for different architectures. Thus, one way to (temporarily) override the default toolchain configuration file (for the current architecture of the project) is to put a modified copy of compile.xml into the project's directory.

The default name/location of the toolchain configuration file can be changed by modifying the bottom entry of the Configuration→Arch+Board … dialog. If the path specified in that entry is not absolute, then the file is sought first in the project's directory and then in the architecture directory of PicOS. If the entry is cleared, then the default setting of compile.xml is assumed. Note that the entire toolchain can be replaced by pointing PIP to an alternative toolchain configuration file. The architecture directory for MSP430 contains a file named compile_ol.xml describing the legacy toolchain as a replacement for the default gcc toolchain maintained by Texas Instruments.

[19] That directory is PICOS/PicOS/*arch*/ where *arch* is either MSP430 or CC13XX.

9 Elvis configuration

Elvis can be configured in many ways and in many aspects, and it is certainly beyond the scope of this note to discuss them all. The editor is equipped with an elaborate help file which can be easily accessed from the program itself. From the viewpoint of PIP, one useful possibility is to assign different color schemes to different file types, which may help in editing/viewing many files at once.[20]

By default, PIP provides elvis with a single (default) color scheme to be used for all files. Whatever other schemes you define, they will be stored in PIP's global configuration file, .piprc in your home directory, and applied globally to all your projects.

To define a new elvis scheme you must still be inside a project (even though the definition is not project-related). This is because the Configuration menu only works if a project is currently open. Click Configuration→Editor schemes. A dialog will pop up. Click New to open another small dialog. Name your scheme (any alphanumeric string will do) and indicate which of the existing schemes will be used for its basis. Initially, there is nothing to choose from except for the default scheme. Click Create to proceed.

In the configuration window that opens next you can assign colors to the various elements (faces) displayed in elvis's windows (you will find their description in the editor's help pages). The first entry, labeled “normal,” is most important: its base color is simply the color used to display (general) text, while the background color refers to the window’s background. The alternate color stands for the second choice for the base color to be used when (according to elvis) it will give a better contrast. Feel free to experiment.

Note that the configuration of a face may be “linked” to some other face. That means that the definition is the same as in the linked-to face, except for the items that are explicitly marked as different in the linking one. The boxes labeled B, I, U, O, stand for bold, italic, underline, and boxed, respectively, and refer to text attributes.

In addition to colors, you can also select the font size for text and, if you know what you are doing, enter explicit, textual, extra configuration commands to be executed by the editor when it starts.

Having completed the definition of a new scheme, hit Done. The scheme will now be available in the menu in the left upper corner of the left (schemes) pane in the previous dialog (which should remain open). You can assign your schemes to file types using the set of widgets in the right (assignment) pane of the schemes dialog. Those widgets are organized into rows, with one row assigning one scheme to some subset of files. The way this is done should be intuitively clear. You select the file type (note that those types correspond to the tree view headers) and where it is coming from (project, system, etc.), with the truly tricky cases resolvable by explicit regular expressions applied to file names. Whenever an editor window is to be opened for some file, the rows will be scanned from top to bottom, and the scheme from the first matching row will be used for the file. If no match is found, the default scheme (which is always defined) will be assumed.

[20] Note that references to C-tags may bring in lots of elvis windows.

10 Multiprogram projects

Section 5 in [mkmk] describes in detail how multiprogram praxes are handled by the command-line tools. Here we show how it all looks from PIP through an easy-to-follow example.

Start by inspecting the project at Apps/EXAMPLES/SIMPLE. This is a single-program project (there's just one app.cc file in the directory) illustrating how to implement simple one-hop RF communication. For a real-life demonstration do this:

1. Build the project, e.g., for the WARSAW (EMSPCC11) board.
2. Upload the Image file into two (WARSAW) nodes.
3. Interface both nodes to COM ports, e.g., via separate FTDI dongles. Start two instances of piter at 9600 bps, mode Direct[21] and make sure they connect to the proper COM ports.
4. Whenever you enter:

   s … ***some text*** …

on one node, you will see the text echoed by the other node.

You can carry out the same exercise virtually in VUEE. For that:

1. Build the VUEE model of the praxis as described in Section 10.3.
2. Run the model with udaemon using data.xml for the VUEE data file.
3. In udaemon, open the UARTs for nodes 0 and 1.
4. Do the same as in point 4 of the previous exercise.

For the multiprogram demo we should come up with at least two different programs to be run by different nodes within the framework of the same praxis. To this end, Apps/EXAMPLES/SIMPLE_TWO contains a derivative of the SIMPLE project with the sender/receiver functionality split between two separate node types, such that a single node can be either a sender or a receiver, but not both.

10.1 Program labels

When you open the SIMPLE_TWO project, you will see under the Sources heading in the tree view two files: app_snd.cc and app_rcv.cc constituting the *roots* of the two programs. The parts “snd” and “rcv” of the file names are called *labels*. They play the role of identifiers of the multiple (two in this case) programs.

The two root “app” files are all there is, i.e., there are no more source files contributing to the project. This of course would be different in a more serious case. The way the source files are attributed to the respective programs is described in Section 3.3 (also in Section 3.5 of [mkmk]). Note that the programs may share files.

When you create a new project, you can choose the multiprogram option in the Project type dialog (see Section 3.1) in which case you will have to provide the labels for the multiple programs. The number of different labels you enter in the dialog will determine the number of programs in the project, and PIP will create that many stub app_xxx.cc files, with xxx replaced with the respective labels. A label can be any alphanumeric string (letters and digits) starting with a letter.

[21] You can run multiple instances of piter from the same PIP session (Execute→Start piter).

Note that to be formally deemed “multiprogram”, in the sense of employing labels, a project doesn't absolutely have to consist of multiple programs. For example, try creating a new project (say a new version of HELLO_WORLD) entering a single label in the Project type dialog and clicking the Multiple programs button. Basically, everything will proceed exactly as before, except that your root (app) file will be labeled, i.e., its name will look like app_xxx.cc where xxx stands for the label you have entered into the Project type dialog. So, the simplest case of “multiple” is in fact “one.” While labeling the single program in a single-program project is allowed, it brings nothing new to what we have already learned about single-program projects. In projects trully consisting of multiple programs, the labeling is necessary as a way of telling the programs apart. From now on, we shall assume that we deal with truly multiple programs (at least two of them).

10.2 Build options for multiprogram projects

Open the SIMPLE_TWO project and click Configuration→Arch+Board. You will see that the Board selection dialog still includes the Multiple checkbox, even though there's no question as to the multiple-program status of the project. This box does make sense. One of its purposes is to tell whether the multiple programs require different devices (boards) and also offering a certain simplification if they don't.

When you uncheck (leave unchecked) the Multiple box, you basically say that all programs of the project are for the same board and they can be compiled as such. There will be only one board to choose in the Board selection dialog, and the choice will apply to all programs. This also implies that all programs share the same local configuration options which, at the project level (as opposed to the board level), are described in the single options.sys file in the project's directory. In the Build menu you will see a single pair of pre-build/build actions applicable to the entire project: the respective action takes care of all programs at the same time (they are all pre-built and subsequently built together). In slightly more technical terms, it means that there is a single Makefile covering all programs.

When you successfully build the project, you will see that the outcome of that action amounts to two Image files named Image_snd and Image_rcv, i.e., the files are tagged with the labels of their programs.

> **Note 10.2.1:** the simplest shortcut to see the full contents of the project's directory is to open an xterm (Execute→XTerm) and execute ls in the terminal. Alternatively, you can click Execute→Run program and type ls in the text widget. Clicking Execute→File Explorer will open a GUI file browser in the project’s directory.
>
> **Note 10.2.2:** each Image file occurs in two versions; this applies to all projects, not only the multiprogram ones (see Section 5).

Now click Configuration→Arch+Board again and check the Multiple box. The shape of the dialog will change allowing you to specify a different board for every program. The project will be built differently now: the Build menu now lists separate pre-build and build actions for each program. Although composite actions (Pre-build all and Build all) are available, they are merely shortcuts for sequences of independent actions carried out individually for each program. Technically, this means that each program gets its own Makefile whose name is tagged with the program's label.

In the Multiple build mode, different programs can specify different local configuration options, which you provide in separate labeled options files, e.g., options_snd.sys and options_rcv.sys

for the project at hand. Those files are not mandatory; if absent, PIP will try to use the global options.sys file for all (both) programs. That file is not mandatory, either: if there's no applicable options file, PIP (mkmk to be exact) will assume that there are no local options. Local options files can be considered a legacy approach to specifying the attributes of the praxis's target hardware, which role has been obsoleted and obviated with the introduction of board descriptions. Unfortunately, we still use them (by tradition) for adjusting or correcting some hardware parameters of the board(s), like the UART rate, the presence or absence of a particular device, etc., while their role should be rightfully confined to specifying configuration parameters directly affecting the praxis programs. In addition to clearly separating concerns, a consistent adherence to this principle will also facilitate board libraries (see Section 11).

> **Note 10.2.3:** one remaining well-defined role of the global options.sys file is to define the configuration of VUEE components (see Section 3.4 of [vuee]).

Even when the Multiple box is checked in the Board selection dialog, you can still select the same board for both (all) programs of the project. This will not affect the independent status of the programs and their independent builds. To sum up:

1. If all programs of a project are for the same board, you have the option of treating them separately or combining them into a single build.
2. If at least some programs require different boards, then you must check the Multiple box in the Board selection dialog, which will result in an independent (individual) build for every program.

Regardless of whether the Multiple box is checked or not, the Image files built in a multiprogram project are always program-specific and they are always tagged with the respective program labels.

10.3 VUEE models of multiprogram praxes

Most of what matters for this leg of the project, we have already said in the previous section. The details, including some of the issues that must be resolved (mostly by PiComp) when turning the set of programs comprising a praxis into a VUEE model, are described in [picomp]. One thing you should remember is that the program will use options.sys to determine the configuration of model components. For example, in SIMPLE_TWO, the file contains a single line looking like this:

```
#define VUEE_LIB_PLUG_NULL
```

which declares that the model needs the NULL plugin (see Section 3.4 of [vuee]).

Make sure that the VUEE model has been configured, i.e., click Configuration→VUEE, check Always run with udaemon, and select data.xml (provided in the project's directory) for Praxis data file (no udaemon geometry file is available in the project's directory, but the model can live without it). Then click Build→VUEE, wait until the compilation completes, and click Execute→VUEE. By looking at data.xml you can tell that node 0 is the sender (snd) and the other node is the receiver (rcv). Open the UARTs of the two nodes. When you type:

```
s … some text …
```

into the UART of node 0, you will see the text echoed by node 1.

11 The library mode

In Section 3.2 we mentioned two build modes for a project, or rather a project's program, because the multiple programs of a multiprogram project can be built in mixed modes, as long as their builds are in fact independent (i.e., the Multiple box in the Board selection menu is checked, see Section 10.2). The library mode is selected by checking another box in the menu, the one that appears under the board name for the given program. If the Multiple box is not checked (and all programs compile as a single build), the library/source mode is selected globally for the entire project.

A library is a precompiled collection of object files. The set of system options for the precompilation, as well as the exact collection of files contributing to the library (i.e., the configuration of system modules) are determined by the board definition. This means that libraries are naturally associated with boards.

The low-level procedures for building and using libraries are described in [mkmk] (Section 4). Within the framework of PIP, a library is built by selecting Create lib for xxxxxx from the Configuration menu where xxxxxx stands for a board name. Such an entry becomes available in the Configuration menu for every board configured into the project unless the editing option for system files is set to Never (see Section 3.5).

Let us return to SIMPLE_TWO for an illustration. We shall reconfigure the project to build in the library mode. Open the Board selection dialog (Configuration→Arch+Board); it doesn't make much difference whether Multiple is checked or not but let us keep it unchecked for now (to make things a little bit simpler). Later you can try to go through basically the same drill with multiple boards. Check the Lib mode box under the board name (which should be saying WARSAW).

With Multiple unchecked, both programs of the project will build together (compiling for the same board). The pre-build action from the Build menu now says Pre-build (WARSAW lib); click it. Unless you have already created a library for the WARSAW board, PIP will complain that no library is available for WARSAW. You can build the library by clicking Configuration→Create lib for WARSAW. Do it now. You will see a bunch of messages in the console pane reminiscent of a regular source build. Finally, assuming that the operation is successful, the console pane will say "–DONE--," and the status line will revert to "Idle." The library has been set up. PIP stores libraries in board directories (they show up in the tree-view pane). The library file is named libpicos.a.

The Configuration menu includes actions for building libraries for all the boards configured into the project.[22] Once built, a library can be rebuilt (just by clicking the respective item in Configuration) but PIP will warn you (and ask for confirmation) if a library for the given board already exists. No dependencies are automatically detected. Normally, libraries must be rebuilt only when some of the system files they incorporate (directly or indirectly) have been modified.

Our next step will be to build the actual project. Now the pre-build action will succeed. Note that both pre-building and building are now much faster than in the source mode because most of the required effort has been already invested into the library.

[22] If the project uses multiple boards, the menu will also show a compound action to build them all at once.

You should remember that, in a library build, any options files present in the project's directory (and sought according to the rules outlined in Section 10.2) have no effect on system parameters, although they can still affect the project programs. And, of course, the VUEE model (which doesn't distinguish between the library and source build modes) still interprets options.sys for the requisite components.

A board description must adhere to certain rules to allow for its library to be created. Those rules are described in Section 4 of [mkmk]. In a nutshell, the file params.sys in the board's directory must specify the list of files to be included in the library.