olsonet Communications

PIP:

an integrated SDK for PicOS

version 0.8

September 2016

Related referenced documents:

[picos]	Programming under PicOS
[installation]	Installation and Quickstart
[picomp]	PiComp: the PicOS Compiler
[vneti]	VNETI: Versatile NETwork Interface
[mkmk]	mkmk: the PicOS Makefile Maker
[serial]	UART Communication via VNETI (TCV)
[emspcc11]	EMSPCC11 brochure
[mspgcc]	mspgcc: A port of the GNU tools to the Texas Instruments MSP430 microcontrollers[1]

1 Introduction

PIP is an attempt at an integrated SDK for PicOS. Before PIP, software development for PicOS involved a collection of command-line tools (mostly Tcl/Tk scripts), most notably mkmk [mkmk] and picomp [picomp], but also genihex, piter [serial], and a few others, including manually invoked command-line utility programs, like gdbproxy and gdb, and independent FET loaders, like Elprotronic FET-Pro430-Lite. Underneath was MSPGCC (the GNU tool chain for MSP430) and Cygwin's emulation of a UNIX environment (which we didn't have to worry about under Linux). Of course, there also was our “platform” consisting of PICOS, SIDE, and VUEE.

When contemplating PIP, I considered some alternatives for its development. At first sight, it would seem a natural choice to use Eclipse for its basis;[2] however, after a brief study, I decided against it. Here are the reasons:

1. Eclipse is a heavy-duty solution, whereas we would prefer something small. The size of its requisite luggage is intimidating. The sheer task of maintenance, e.g., keeping track of upgrades, would be very absorbing.
2. Eclipse is Java-based and, no matter how much one may want to disguise that fact, Java-oriented. Its power looks much less impressive if Java is totally of no interest to the developer (as it happens in our case).
3. Software development under PicOS is not exactly a straightforward instance of the generic case of software development (in Java or C++). We have our own objectives and our own (idiosyncratic) targets, e.g., VUEE models (with their data files, including udaemon's geometry data), multiple clones of the same uploadable Image differing by the node number, and our own organization of the “libraries” with the PicOS source code being a special (shared) part of all user projects. In other words, our SDK is conceptually and organizationally quite different from IAR or CCS (which are the two most popular commercial systems for programming MSP430-based microcontrollers).

Point 3 makes PIP sufficiently different from the “standard” case to cast serious doubts on the usefulness of Eclipse. All those idiosyncrasies of our setup would require special handling, meaning that on top of the other problems, the amount of work needed to get

[1] By Steve Underwood.

[2] There exists an Eclipse-based SDK for MSPGCC (see http://homepage.hispeed.ch/py430/mspgcc/). I cannot say anything about its quality, except that it is certainly not geared for PicOS or VUEE.

something truly useful (as opposed to superficially flashy) within the Eclipse framework would be quite nontrivial.

Consequently, I decided instead to hack something in Tcl/Tk, which has been our workhouse from day one. What I have managed to procure has been surprisingly effective and useful from the the first moment of its internal release. For a newcomer it simplifies many things obviating the need to read our documentation, which is not always friendly. It makes the so-called quick start quite a bit quicker than it used to be.

This document does not describe everything about PIP. Being a GUI program, PIP needs no detailed explanation of every single item in every menu. So this is merely an introduction. Some items that may be difficult to grasp at first sight, are covered in more depth.

2 PIP under Windows

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 mostly use Windows + Cygwin. 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 (DOS) format in all those situations when they have to 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)[3] in the Windows context, i.e., formally it needed no Cygwin to live. That created some confusion for at least two reasons:

1. File paths used by PIP were in DOS 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 remains an option (and I actually prefer it, for the considerably better looks of its GUI, to the “more natural” and default Cygwin version of the environment). 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.

2.1 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. Owing to the fact that the windows generated 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 forces 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 have to explicitly hide all native windows (resuming the X11 pane), and vice versa, which is extremely annoying.

When PIP runs under the Cygwin's native version of Tcl/Tk (which is the default), 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 more comfortably run most things under X11 Windows, also in windowed mode. However, every once in a while PIP may want to invoke some external program executing solely under Windows (the Elprotronic

[3] See www.activestate.com.

FET programmer is one example that immediately comes to mind). So it seems that the multiwindow mode of X is still a more convenient choice. Personally, I find the multiwindow mode generally much better (friendlier) than the other mode, although for quite some time in the past I would stick to the windowed mode in my hopeless efforts to pretend that there was no Windows at all.

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

As you can see, 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.

With the X11 server operating in the multiwindow mode, most of PIP's kludgy nature will be hidden from your view. One little problem (which I couldn't do much about) is that the text editor's windows (under X11) cannot sensibly negotiate their way up through other windows, so they may open (partially) obscured (this problem doesn't occur in windowed mode). This is because in the multiwindow mode the whole set of editor windows is considered second-class compared to Windows native windows.

2.2 The "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 (and thus, at least to some people, appear nicer). Of course, the ActiveState Tcl/Tk package has to be installed for that.

Note that, even in the 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.

3 Invoking PIP

Just type:

```
pip
```

in an xterm. Alternatively, you can go:

```
pip -D
```

(capital D), if you want to see debug messages produced by PIP on its standard output. Use this if you detect a (hopefully replicable) problem that you would like to report.

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. In particular, 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 half way through a compilation is obviously not recommended.

4 A walk through a simple project

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

We shall create a simple project from scratch and go through all the essential steps of its completion. For the praxis,[4] 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. Besides, we will end up with something slightly more elaborate than the ready version.

4.1 Creating a new project

Start PIP, click File → New project and choose a directory. This will be a new directory, which should belong to the tree rooted at PICOS/Apps. 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 4.1.1:** 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-style file browser.
>
> **Note 4.1.2:** do not use file/directory names including spaces (PIP won't let you use them, anyway, but it cannot prevent you from creating ones spuriously under the standard open file dialog).[5] 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.

[4] This is our name for a PicOS application.

[5] 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.

When you select/type-in the directory name and close the dialog, a new (project type) dialog will pop up: you have to decide whether the project is for a single program or for multiple programs. A multiple-program project is for those cases where the network needs nodes of different types, i.e., running different programs, which nonetheless belong to the same application. In a situation like that, you have to assign labels (alphanumeric keywords) identifying the different programs (see Section 6). As our simple project is going to comprise a single program, just press the Single program button and the dialog will go away.

The title of PIP's root window will now be showing something like “PIP 0.8, project HELLO_WORLD”. The left pane will present the initial list of the (relevant) files in the folder belonging to the project. For a new project, PIP puts there two files acting as stubs (to be filled out by you), 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. Actually, the second file, options.sys, is optional, but it is probably a good idea to keep it around, even if it is empty.

Files in the left pane of PIP's main window (we shall be calling it the *tree view*) are organized into five groups named **Headers** (.h files), **Sources** (.cc files), **Options** (options files), **XMLData** (data files), and **Scripts** (e.g., OSS scripts). Later, you will probably 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 in the tree view (but only if the project's options make system files viewable, see Section 4.5, esp. Note 4.5.1).

4.2 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. When you click on the Build menu, you see that the only options available are for VUEE, i.e., for the emulator. 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 → CPU+Board which will produce a small Board selection dialog. Don't touch the MSP430 menu (this is our CPU type) and don't check the Lib mode box, but click on the button under Board (initially showing ---) and select WARSAW → WARSAW (or whatever your board type may be). Note that the menu entry for this selection is two-level. This is because there are several subtypes of the WARSAW board [emspcc11]: you have to select the vanilla one (the first one from the second-level list). Then click Done. The tree view will now show (at the bottom of the previous set) a new entry (with a distinctive gray background) saying <WARSAW>. 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. 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 in a traditional setup corresponds to configuring the program for the specific environment, and the actual compilation. The outcome of the first step is a Makefile[6] to be used in the second step.

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

So when you click Pre build (WARSAW src), 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 summoned 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 show up to the right. This means that some program is now running and (possibly) writing its output to the console. When mkmk has finished, the text --DONE-- will appear at the end of the console's output and the status line will say Idle again.

> **Note 4.2.1:** 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 3).
>
> **Note 4.2.2:** 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 really 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 (typically) 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: Clean (full) and Clean (soft).[7] The first button removes all files that can be 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 cleaning doesn't destroy 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 user output from this action, including any 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 goal by clicking Build (make) directly. If no Makefile is present when a build is requested, PIP will carry out the pre-build action automatically, effectively emulating your clicking the Pre-build entry in the menu. 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. And, of course, pre-building removes all output of a previous build.

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

4.3 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. So 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 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.[8] 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 is being edited that 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. Needless to say,

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

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.

4.4 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 (just 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");

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

Now close both editor windows, pre-build, and then build.[9] This time you will get two errors (note that we didn't fix 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 have to make sure that the

[9] Note that this time you have to pre-build, because you have added a new file to the project.

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 and also insert there a header of rotate_leds, i.e.,

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

Suppose that you want to have a look at rotate_leds before fixing the problem (and your program is not as trivial as our educational example). Now, 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 at relevant stages re-builds) 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 (assuming that those things are always easy to see in their usage context). 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.

4.5 Access to system files

Click Configuration → Options. You will see a dialog defining three parameters associated with your project:

- the maximum number of lines stored in the console pane in PIP's main window (defaulting to 1000); any lines exceeding this limit will be discarded off the pane's top
- 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 system model) you want to see

Note that VUEE is a separate “system” with its own source files implementing SIDE models of the PicOS stuff and the network environment. If you are running 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 4.6).

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 also 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 (from inside a project). 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, as long as 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, which are stored globally (for the user).

> **Note 4.5.1:** 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 system files at all, PIP will hide the board definition(s) from the view.

The C-tags database for your project's 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 actually 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 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).

4.6 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 figure out what that file might be would be to double click on the respective function call 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/Libs/Lib/ser.h
```

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

[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.

so you have to 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 error locations produced by the compiler. When you double-click on such a header, and the current option settings (Section 4.5) 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 collection of widgets. 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.

You could narrow down the search even more by adding an opening parenthesis after ser_out in the String field. When you try it (together with h$ in FN Pat), you will get exactly one match. 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]

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/
```

stands for 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 will be searched

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

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

Note that in the second case the project must have been pre-built (see Section 4.5) for the PicOS files to be searchable (only then can PIP know the exact pool of files to search).[13] This does not apply to the 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); searching 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 refers to the files that *will not* be searched.

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.

4.7 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 that exercise, we shall assume that your hardware connection to the device consists of the FTDI TTL232R USB UART dongle and the MSP-FET430UIF programmer, both devices connected to the PC 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 COM device of the FTDI dongle. If you don't know it, hit the Scan button; then the device menu will only show those COM devices that respond. Note that with PIP running under the Cygwin version of Tcl, 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). The dongle is usually easy to spot as something new and different from what you are used to. For subsequent sessions, the same dongle will tend to map to the same COM device.[15] 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.

[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.

[15] This isn't true on Linux where the numbering of ttyUSBx devices reflects the order of their connection.

Click Configuration → Loaders to select the loader program. With MSP-FET430UIF, and under Cygwin, your immediate options include MSPDebug and Elprotronic FET-Pro430-Lite. The Elprotronic loader can be downloaded (for free) from https://www.elprotronic.com/. MSPDebug (by Daniel Beer http://dlbeer.co.nz/mspdebug/) comes included with the MSPGCC compiler (see [installation]). On Linux, MSPDebug is the only option unless you configure some other loader in the Command line section (see below).

For now, choose MSPDebug in the Loader configuration window (which will work on Windows + Cygwin as well as on Linux), make sure that FET device for MSPDebug is set to Auto, Driver is set to tilib, the Allow firmware update box is checked, and GDB connection port is None. Click Execute → Upload image in PIP. MSPDebug may want to update the firmware in the programmer (MSP-FET430UIF) first. If so, this will only happen the first time around (as described at a similar exercise in [installation]). Eventually, it should put 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.

Note that if the praxis consisted of multiple programs, i.e., multiple image files for upload, PIP would ask you, in a separate window, to select the actual file to be submitted to the loader.

Now try the Elprotronic loader (assuming you are on Windows and the program has been installed). Select it in the Loader configuration window instead of MSPDebug. The configuration is simpler in this case, because there is only one parameter, Path to the program's executable, which normally (for a standard installation of the loader) should be left at Auto. Now, when you click Execute → Upload image, the loader will open its own GUI window. The file to upload (near the Open Code File button) will be already filled in with Image.a43, which is the HEX version of the uploadable image created from the project. Now press AUTO PROG which, if everything is connected as needed, should cause the image file to be flashed into the device.

Note that the window of FET-Pro430-lite does not go away after the program has been loaded. You can use it to flash other boards and also control (reset) the currently connected board. You can close the loader directly (from its own window) or from PIP's Execute menu.

> **Note 4.7.1:** when you often switch between the two loaders, they may have different ideas as to the firmware version that should be written into the FET programmer (MSP-FET430UIF), and its constant reprogramming may be annoying and time consuming. The firmware version is determined by the version of the MSP430 interface library (provided by Texas Instruments) used by the two programs. The library used by MSPDebug can be found in C:\mspgcc\bin (file MSP430.dll) while the one used by the Elprotronic loader is located in the program's installation directory (normally C:\Program Files (x86)\Elprotronic\MSP430\FET-Pro430) and named MSP430-TI.dll. You can copy one onto the other thus unifying them and making the two loaders agree on the programmer's firmware version. You probably won't err when you assume that the larger file represents the newer version of the library.

Generally, a real-life (as opposed to VUEE) compilation of a program results in two versions of the same image: an ELF version (file name without an extension, e.g., Image) and a HEX-format file (extension .a43). The root name of an image file always begins with Image; if you have multiple programs, or if you have parameterized images (Section 4.8), the name may include other components, but its extension can only be .a43 (or absent) and it always clearly discriminates between the two content types.

Some flash loaders, which you may want to use externally (and which PIP knows nothing about), may expect other file extensions. Most of the ones I have seen accept the HEX file format, but some expect the extension to be .hex or .txt. In such a case, you will have to copy or rename the respective .a43 file.

There exist other loaders (for Windows as well as for Linux) offering their specific interfaces which may be easy to configure with PIP taking advantage of the Command line option in PIP's Loader configuration window. Such a loader is configured by specifying the path to the loader's executable and the argument string to be passed to the program. The sequence %f occurring anywhere in the argument string will be replaced by the file name with the flash image. 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 one with the extension .hex before submitting it to the program.

Note that potentially quite complicated cases can be handled by custom wrapper scripts reducing the complexity of passing parameters to the loader. If you have to use a loader with its own GUI and (unlike the Elprotronic loader) admitting no configuration files that would allow a script to sneak in parameters before its invocation, then you can just put the loader's path into the Command line section of the configuration window and specify the parameters (including the file to upload) by talking directly to the loader. This will not be much of a shortcut, however; more like a reminder what kind of a loader should be used to upload the images generated by this particular project.

4.8 Parameterizing uploaded images

Quite often different nodes of the same network have to 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 is the Node Id (e.g., used as node address in TARP). 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 long-word (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 simple idea is that the original (input, source) 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 (it is not a good idea to use 0 or 1, but something much less likely to occur by chance). By default, the generator assumes that the first value is 0xBACADEAD and the second one is 0xBACADEAF. The first value is a standard tag used by PicOS to refer to the 32-bit identifier known as the Host Id (or HID) and declared as:

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

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

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

presetting it to the discernible value.

The topmost widget in the generator dialog window is a menu to select the image file format which can be IHEX (for the .a43 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 template) and the other describing the pattern to be applied for naming the files containing the clone 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 file name includes the sequence “_nnnn” (literally: 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) of 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 “second value” and APP Value is its replacement. If APP Value is empty (which it is by default), the second value is neither sought in the source image nor replaced.

The field labeled From specifies the starting value for the lower half of host_id and How many tells how many clones are to be generated. The minimum initial value is 1 (note that it 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 program will object as well (it wouldn't know which occurrence to replace). 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 create 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 (the Output field) with a source file (the Input field). Its primary purpose is to verify whether an image has been produced by cloning a given source and to show the identifying values of the clone with respect to the source image.

4.9 Debugging

It is possible to debug programs running in real devices with gdb (I mean msp430-gdb). The interface involves MSPDebug acting as a proxy to msp430-gdb. I don't want to go into too many details (or give elaborate examples), mostly because I practically never use gdb. The only cases when I actually use it involve finding bugs in VUEE or VUEE models, so it is *not* for debugging programs running in real-life microcontrollers. One of the ideas behind VUEE is to render debugging such programs unnecessary.

Here is the way to do it in a semi-automatic sort of way:

1. Select MSPDebug in the Loader configuration window. The only difference with respect to a standard case of uploading is to select a GDB connection port, e.g., 2000.
2. Click Execute → Upload image. This will not upload the image to the board. Instead, MSPDebug will start as a debug proxy and PIP will invoke msp430-gdb in a separate xterm window specifying the selected image file as its argument.

Now you can upload the image manually from msp430-gdb and, possibly, do some debugging. For example, these commands:

```
monitor erase all
load Image
```

upload the image file to the device. Then, when you execute

```
continue
```

the program will start and run until a breakpoint. Refer to [mspgcc] for more information.

4.10 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 have to worry about 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) 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 from for the same effect.

Navigate to EXAMPLES/INTRO/HELLO_WORLD/data.xml and “open” that file. It will be copied to your project and appear under XMLData (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 model. Check Do not propagate board config to VUEE, 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: 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. A WARSAW node is equipped with radio, so the model would complain about the absence of a radio channel for the network.

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. You will see the “Hello world” message, as before, except that now it is coming from a virtual device. To see the LEDs, select LEDS from the menu in udaemon's window, the one 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, as many times as you please.

It is all much more impressive with a 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.

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.

4.11 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 4.9). 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 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 may want to enter:

```
run data.xml +
```

instead. The second argument (standing for the output file for the VUEE model) indicates that the output should be written to the standard error (rather than the default standard output) with the net effect of flushing the output on every line (so you can see it right away). This problem is specific to Cygwin/Tcl and results from one of the never ending list of obnoxious mismatches. Another problem is that Control-C doesn't seem to work for gdb under Cygwin (I distinctly remember that it used to work fine, or maybe it still does on every second day?).

You can abort any program running in the console with Execute → Abort (or Build → Abort, both choices trigger exactly 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 → XTerm and simply open an xterm client in the project's directory. The xterm is completely detached from PIP; in particular, it won't go away when PIP exits.

5 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 lots of files at once.[16]

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 be inside a project. 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 applies to the window 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.

6 Multiprogram projects

Section 5 in [mkmk] describes 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.

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

3. Interface both nodes to COM ports, e.g., via separate FTDI dongles. Start two instances of piter at 9600 bps, mode Direct[17] 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 4.10.
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 have to 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.

6.1 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 is 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 4.4 (and also in Section 3.4 of [mkmk]). Note that the programs may share some files.

When you create a new project, you can choose the multiprogram option in the Project type dialog (see Section 4.1) in which case you will have to provide the labels for the multiple programs. The number of different labels you enter in the Project type 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.

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” in this case. 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 actually consisting of multiple programs, the labeling is necessary as a way of telling the programs apart. From now on, we shall assume that we *actually* deal with multiple programs (so “multiple” means at least two).

6.2 Build options for multiprogram projects

Open the SIMPLE_TWO project and click Configuration → CPU + 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. Its one

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

purpose 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 more technical terms, it means that there is a single Makefile for 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 6.2.1:** the simplest shortcut to see the full contents of the project's directory is to open an xterm (Execute → XTerm) and execute ls. Alternatively, you can click Execute → Run program and enter ls into the text widget.
>
> **Note 6.2.2:** each Image file occurs in two versions; this applies to all projects, not only the multiprogram ones (see Section 4.7).

Now click Configuration → CPU + Board again and check the Multiple box. The shape of the dialog will change allowing you to specify multiple boards on a per program basis. The project will be built differently now: the Build menu 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 which, as one can easily guess, 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 7).

> **Note 6.2.3:** 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 independent builds for all programs.

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.

6.3 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 for Praxis data file (no udaemon geometry file is provided 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.

One little thing you may want to know is that the two domains of the praxis, i.e., VUEE and the real world, are not entirely disjoint. You may compile and run a VUEE model without specifying anything about the real hardware (boards) of the praxis program(s), but then the model will have no way of knowing what those boards might be. Section 3.2 of [mkmk] discusses some board-related symbols available to a program being compiled for a real board that the program may reference to see what it is being (or has been) compiled for. If your model wants to have access to the same symbols, specifically `BOARD_TYPE` and `SYSVER_B`, make sure to define the board(s) for real-life compilation before building the VUEE model. Then PIP will pass the board information to the model as well (see also [picomp], Section “Parameterization issues for multiprogram praxes”).

7 The library mode

In Section 4.2 we mentioned two build modes for a project, or rather a project's program, because the multiple programs of a multiprogram project can be build in mixed modes, as long as their builds are in fact independent (i.e., the Mulltiple box in the Board selection menu is checked – see Section 6.2). The library mode is selected by checking another box in the Board selection 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 such a 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 (and rather strongly) associated with boards.

By invoking mkmk manually (see Section 4 in [mkmk]), one can create libraries that can be additionally parameterized by local options (supplementing or modifying those associated with the definitions of the corresponding boards). Within the framework of PIP you can create and subsequently use only “pure” libraries, i.e., ones whose shape is solely determined by the configuration of their boards. As mentioned in Section 6.2, the whole idea of augmenting (or correcting) board definitions by local options is probably flawed and should be avoided in serious (production) projects.

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 → CPU + 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 such a library by clicking Configuration → Create lib for WARSAW. Do that. You will see a bunch of messages in the console pane reminiscent of a two-stage build of a regular program: there will be a pre-build stage followed by a compilation. Finally, assuming that the operation is successful, the console pane will say –LIBRARY CREATED--, and the status line will revert to idle. The library has been set up. PIP stores libraries in directory LIBRARIES of the PICOS tree. This directory is created on the first library build, if it doesn't exist. The libraries are named after their boards; thus, the operation described above will yield subdirectory WARSAW in LIBRARIES containing the library for the WARSAW board.

The Configuration menu includes actions for building libraries for all the boards configured into the project.[18] 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 have to be rebuilt only when some of the system files they incorporate (directly or indirectly) have been modified.

Our next step will be to actually build the 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.

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 6.2) 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.

8 Comments

People used to accomplishing their goals solely by clicking buttons will be disappointed by the fact that PIP leaves considerably more to textual input than “more professional” solutions. For example, other SDKs would let you add files to the build by clicking on them rather than forcing you to insert special comments into other files (see page 10). Personally, I find it more natural to reflect the fact that something is needed by my program within the specific fragment of code that depends on the requisite item than to rely on a mark in some obscure internal description of the project. Traditionally, such information has been handled quite well textually through header files, and it is still difficult to beat its “expressive power” by clicking buttons.

Another thing that PIP does little about is hardware description. Traditionally, we have been storing hardware (board) descriptions in a special directory inside the PICOS tree (see PICOS/PicOS/MSP430/BOARDS) as collections of header files. While it is conceivable to visualize those descriptions better, e.g., turning them into a bunch of selections, entry boxes, radio buttons, and so on, I wouldn't want to loose in the process the convenience of being able to keep there comments (sometimes quite elaborate) as well as miscellaneous pieces

[18] If the project uses multiple boards, there will be also a compound action to build them all at once.

of code that fit there better than anywhere else. So I still believe that nothing beats the simplicity of being able to mix so much “multimodal” information in a short file that you can see all at once in an editor window.

One thing that probably would be useful is a special collection of simple widgets for editing and navigating board descriptions. You can do it now by using the buttons in the search window, but perhaps something specifically addressing this particular end would not be completely out of place.