olsonet Communications

PIP:

an integrated SDK for PicOS

version 0.4

December 2011

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 and picomp, but also genihex, piter, and a few others, including manually invoked command-line utility programs, like gdbproxy and gdb, and independent FET programmers/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;[1] 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 you may want to disguise that fact, Java-oriented. A lot of 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 simple 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.

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

Consequently, I decided to hack instead something in Tcl/Tk, which has been our workhouse since day one. What we have now can be considered a starting point which, however, is already quite useful and effective. Especially for a newcomer it simplifies many things obviating the need to read our documentation, which is not always friendly. It should make the quick start a bit quicker than it used to be.

2 Using X-Windows under Cygwin

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. Its main window and the dialogs triggered by its menus operate exclusively in the Windows context, i.e., they formally need no Cygwin to live. Its text editor, however, needs X, which runs under Cygwin. Although other X-Windows systems are available for Windows, the X server that comes with Cygwin is a natural choice, because PIP needs Cygwin (its set of utilities) for other things as well.

In consequence of this kludgy setup, Windows windows must coexist with X-Windows windows within the context of something that we would like to call a single application. To make most sense out of this, you should run the X-Windows server in the *multiwindow* mode, i.e., without the background window covering the entire screen (which is known as the *windowed* mode). This is because, in the windowed mode, whenever you want to see an

[1] 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.

editor window, you will have to explicitly hide PIP's windows, and vice versa, which is going to be extremely annoying.

These days 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 effort to conceal the fact that there's Windows underneath.

The recommended way to start the X server in the multiwindow mode is to double click Cygwin-X → XWin Server in the Windows startup menu (needless to say, you can create a convenient Desktop shortcut for it). The server reads a 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 task bar, which you can right-click to open xterm windows and stuff, as well as to terminate the server. There is no need to access Cygwin utilities in any other way, so you can forget about the Cygwin terminal for one thing.

Here is my personal and trivial version of .XWinrc (which is not far from the default one). I have removed the indirection required to access my most popular application, i.e., the xterm client, under the default setup, which you can find (for useful comments and reference) at /etc/X11/system.XWinrc.

```
# XWin Server Resource File (simplified by PG)
menu root {
       "Reload .XWinrc"    RELOAD
       Separator
       xterm        exec   "xterm"
       Separator
}
RootMenu root
SilentExit
DEBUG "Done parsing the configuration file..."
```

With the X 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 editor's windows cannot sensibly negotiate their way up through PIP's (Windows) windows, so they may open (partially) obscured. This is because the whole set of editor windows is considered second-class compared to Windows windows.

3 Invoking PIP

Just type:

```
pip
```

in an xterm. Note that it must be an xterm (or possibly some other X client), because PIP must know the identity of the X display. Strictly speaking, it is elvis, PIP's editor, who needs to know that identity, but elvis is invoked from PIP, so it boils down to the same thing.

In fact, this may be a bit confusing, because PIP will start without the X server (e.g., when invoked from a simple Cygwin terminal). It will even work and do things for you for as long as you don't try to edit a file, in which case PIP (or rather elvis) will fail complaining that it cannot talk to an X server.

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.

4 A walk through a simple project

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

We shall create a simple project from scratch and go through all the essential steps of its completion. For the praxis,[2] we shall use the "Hello World" program from Section 2.1 of PicOS.pdf. Even though this praxis already has an implementation (to be found in Apps/EXAMPLES/INTRO/HELLO_WORLD), we shall do it again. Besides, we will end up with something a bit more complicated than the old 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_WORD.

> **Note 4.1.1:** if you are on Windows/Cygwin, the dialog shows a Make New Folder button. On Linux, there is no button, but you can just type the directory name at the end of the path shown at the bottom of the dialog. The Windows dialog is confusing, so be careful. I may get to programming a better (custom) version of this dialog some day.
>
> **Note 4.1.2:** do not use file/directory names including spaces (PIP won't let you, anyway). They are considered illegal by PIP, because they confuse some of our underlying tools. This may be fixed at some point, but it isn't a high priority item.

When you select the directory 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 this, you have to assign labels (alphanumeric keywords) identifying the different programs (see section 5 of mkmk.pdf if you are curious about details). 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 0.4, project HELLO_WORLD". The left pane will present the initial list of all the relevant files 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. The second file, options.sys, is in fact optional, but it is a good habit to keep it around, even if it is empty.

Files in the left pane of PIP's main window (we shall call it the *tree view*) are organized into four groups named **Headers** (.h files), **Sources** (.cc files), **Options** (options files), **XMLData** (data files). Later, you will probably learn that there can be more than one options.sys file (when the praxis is multi-program). The data files (their names mostly ending with .xml) pertain to the VUEE model of the praxis.

4.2 Compiling for real hardware

The project can be compiled right away and even loaded into a board, although the latter is pointless, as the program literally 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. This is because you haven't assigned a board to the project yet, so PIP doesn't know how to

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

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), but click on on the button under Board and select WARSAW (or whatever your board type may be). Then click Done. When you look at the Build menu again, you will see that it offers two more choices appearing at the very top, with the first choice being enabled (i.e., clickable). It says Pre build (mkmk WARSAW), assuming that WARSAW is indeed your board type.

To fully understand how a loadable image of your program is created, you should read mkmk.pdf. For now, suffice it to say that the operation is carried out in two steps: pre-building, which in a conventional setup corresponds to configuring the program for the specific environment, and the actual compilation. The outcome of the first step is a Makefile[3] to be used in the second step.

So when you select Pre build (mkmk WARSAW), you will configure your project for compilation. You will see in the right (console) pane of PIP's main window a piece of text produced by mkmk, which has been called 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.
>
> **Note 4.2.2:** a program running in the console can be aborted by clicking 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.

One thing you should be aware of from the very beginning (this is also explained in mkmk.pdf) is that the compilation of a PicOS praxis also involves the compilation of all system (PicOS) files needed by its program(s), including the kernel source code. This is because those files are heavily parametrized 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, or when you pre-build it, which action also removes all previously compiled object files.

You will notice that the Build menu shown two cleaning buttons: Clean (full) and Clean (soft).[4] 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 typical 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. Note that following the successful pre-building, the second choice from the Build menu, Build (make), becomes enabled. When you click it, 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.

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

[4]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 board. 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[5] which is a moderately popular VI-compatible, X-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 with a single empty state. Edit it out to look to 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 did pre-build 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

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 comforting a handful of idiots. 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, you can save the file 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).

[5]Written with a small 'e' reserving the capital letter for the King.

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 amounting to 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. Let us forget about the problem for 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 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. This time you will get two errors (note that we didn't fix the previous one yet, so it is bound to show 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 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 Image be explicitly referenced from other files that are already marked for compilation (see Section 3.4 in mkmk.pdf). 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
```

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 of 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 window'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 below).
>
> **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 purely 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).

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 the system files, the situation is a bit more complicated (and is different for PicOS and for VUEE).

For PicOS, only those system files that your project actually needs are C-tagged. Their set can only be known after you have pre-built the project. Consequently, no PicOS files are accessible via C-tags clicks if the project hasn't been pre-built yet, or after you have fully cleaned it (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).

For VUEE, assuming the C-tags option is on, the C-tags are computed once for all, for the complete set of 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 the line showing up in the console pane to tell you: “Tag ser_out not found”. 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 again 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
```

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 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 width is determined by the Bracket field from the lower line 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[6]

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

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). This does not apply to the VUEE files (if the VUEE box is checked), which are always searched *all*.[7]

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).

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 marks the files that *will not* be searched.

[6]See http://www.tcl.tk/man/tcl8.5/TclCmd/re_syntax.htm for a detailed description of regular expressions in Tcl.

[7]This reflects the fact that all those files are compiled into any 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. 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.

4.7 Uploading programs into boards

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 board.

For that exercise, we shall assume that your hardware connection to the board consists of the FTDI TTL232R USB UART dongle and the MSP-FET430UIF programmer, both devices connected to the PC via USB cables. All the requisite drivers are present as per Installation.pdf.

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, once the board gets to running the program.[8] 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. 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.[9] Select the speed of 9600 bps and 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 defaults.

With MSP-FET430UIF, and under Cygwin, you have two options regarding the loader program.[10] In PIP's main window, click Configuration → Loaders. You can choose between Elprotronic FET-Pro430-Lite and msp430-gdb (this is actually a debugger also providing loader functionality) communicating with the board via msp430-gdbproxy. Let start from the second option. Select Use for msp430-gdb and make sure that FET430UIF is shown in the menu as the programmer device. Then click Done.

Now click Execute → Upload image. Note that this item only becomes enabled (clickable) after a successful compilation (an Image file is present in the project's directory). A window will pop up with two panes. The left pane shows the output of msp430-gdbproxy, while the left pane is for msp430-gdb, which is initially not active. If the bottom of the left pane says something like PROXY CONNECTION ESTABLISHED, it means that msp430-gdbproxy has successfully connected to the device and is ready to receive commands from msp430-gdb. Otherwise, something is wrong and, usually, some hints regarding the reason are displayed in the pane.

One possible reason for failure at this stage is the wrong firmware in your programmer, which is quite likely, if you have never before used the device with msp430-gdbproxy. In such a case a pertinent message should show up in the left pane. Click the Update FET button and wait for a few seconds. If everything works as desired, you should see a bunch of

[8] The program is documented in Serial.pdf.

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

[10] On Linux, msp430-gdb + msp430-gdbproxy is your only option.

messages in the left pane indicating progress in feeding new firmware into the programmer. Wait until you see REPROGRAMMING COMPLETE and then hit the Restart button.

> **Note 4.7.1:** never try to reprogram Olimex's JTAG TINY programmer. When you do that, you will likely damage the device.

Once the proxy has connected to the board, the next step is to choose the Image file to upload. There is not much choice in our simple case (there is only one Image file), but sometimes (e.g., for multiprogram projects) this step may be nontrivial. The menu at the bottom of the upper right column of buttons lists all Image files that PIP sees as possible candidates for upload. They only file present there in our case is named Image. This is the ELF[11] output produced by the linker at the end of build (compilation).

Push the Connect button to start msp430-gdb for the (selected) Image file. The right pane should come to life displaying the initial output from msp430-gdb. If everything works fine, you should see the text GDB CONNECTED TO DEVICE at the bottom. Also, the three buttons under Connect should become enabled. Click Load. After a few seconds the left pane should show a bunch of "memory write" messages from msp430-gdbproxy. Eventually you should see the text PROGRAM LOADED SUCCESSFULLY at the bottom of the right pane. The program has been loaded. Click Run to reset the board and start the program. In piter's window you should see a standard PicOS header (written to the UART after every reset), followed by "Hello world". The three LEDs will begin their rotation switching every second. You can now disconnect the board from the programmer, but you may want to close the loader window first, so msp430-gdbproxy will not feel offended.

Note that the right pane of the loader window gives you full (manual) access to gdb with the three buttons (Load, Run, Stop) providing merely shortcuts for the most typical operations. The input field at the bottom can be used to enter directly any command. In particular, you can carry out the (previous) load procedure with these commands:

```
monitor erase all
load Image
continue
```

You can also set breakpoints and debug the program. The document by Steve Underwood, known as mspgcc.pdf in our set, provides a few useful hints.

4.8 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 have to worry about is a data file to the model, which describes such things as the population of nodes, their hardware configuration, their spatial distribution, the parameters 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 through VUEE.

Click Configuration → VUEE to configure the model. Check Always run with udaemon, and select data.xml for Praxis data file. No need to touch the remaining two selections. Click Done.

[11] Executable and Linkable Format. This is the standard format of object modules and executables produced by the GNU linker.

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 Id 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 board. To see the LEDs, select LEDS from the menu in udaemon's window and click Connect again, this time to connect to the node's LEDS module.

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.pdf for details.

4.9 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. Note that debugging a program running in real hardware can be done from msp430-gdb (used as the loader), although personally I have practically never resorted to that method. There is something about programming in PicOS that makes debugger-assisted program analysis considerably less appealing compared to other systems. In the vast majority of cases the problem is either conceptual (and the debugger cannot possibly help), or it can be fixed much faster by a simple inspection of code. Even for VUEE, most of the situations where the debugger does help involve errors in the VUEE emulator rather than in your praxis code (so it more for VUEE development than anything else). Then, the debugger's role usually reduces to pointing at the line that triggers the segmentation violation, or whatever the exception happens to be.

To see how the debugger 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, hit 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/Tcl8.5 and results from one of the never ending list of obnoxious mismatches. Another problem is the 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, which is 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 terminal 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.[12]

By default, elvis comes 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, however. 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 determine 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 is pretty obvious. You select the file type 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 the 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 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 8). 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

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

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 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.