Pawel Gburzynski

PiComp:

the PicOS Compiler

version 0.5

July 2021

Related referenced documents:

[picos]	Programming under PicOS
[pip]	PIP: an integrated SDK for PicOS
[m]	VNETI: Versatile NETwork Interface
[vuee]	VUE[2]: the Virtual Underlay Execution Engine
[mkmk]	mkmk: the PicOS Makefile Maker
[mspgcc]	mspgcc: A port of the GNU tools to the Texas Instruments MSP430 microcontrollers[1]
[armgcc]	Composite documentation of the ARM gcc compiler available with the installation (e.g., from https://developer.arm.com/)
[installation]	Installation and quickstart

Introduction

PiComp performs two types of jobs:

1. Preprocessing source files in "enhanced" PicOS language (see below), such that they can be handled by the (standard) C compiler (say, *gcc* in *mspgcc*).

2. Transforming PicOS praxes (possibly multiprogram praxes) into VUEE models [vuee].

Regarding the first type of job, we would like to have a more convenient way of expressing certain PicOS constructs than what can be achieved by relying solely on C preprocessor macros. Old versions of some of those constructs, especially process (strand/thread) declarations, look weird. In addition to resolving those problems, a more powerful pre-compiler will allow us to think about new syntactically viable constructs that could be added to the PicOS language to facilitate programming in its true spirit. While, at this time, the contribution of PiComp to enhancing the programming language of PicOS may appear moderate, we may view it as the first step. It may be comforting to know that, should we come up with something more creative, the path to adding new (possibly non-trivial) constructs to the language is essentially open.

For the second role of PiComp, the manual effort needed to maintain VUEE compatibility of PicOS praxes has been quite serious and irritating, despite our attempts to simplify the rules and recommendations. There has been a rather destructive tradeoff between the ease of maintenance (of VUEE-compliant code) and the degree of structural finesse of the program that one was wiling to indulge in (having to keep in mind the complexity of maintenance). While the most recent set of rules and recommendations for achieving VUEE compliance "manually" may not be excessively complicated, those rules nonetheless force one to organize the program into a rigid collection of files while paying close attention to several counter-intuitive and messy details (like the multiple *contexts* of a variable declaration). The easiest way to avoid mistakes is to minimize the number of all those tricky places that must be carefully checked/updated when modifying the

[1] By Steve Underwood.

praxis, which in turn stimulates restrictive simplicity, like making all variables global and keeping them all in one place.

In this context, the role of PiComp is to transform the source program in PicOS mechanically into a set of files that can be then comfortably (and automatically) processed by the VUEE/SMURPH compiler. The original source program doesn't have to be especially prepared for that task, except for the avoidance of some obscure and basically irrelevant constructs, which reasonable programmers would statistically never use anyway. While I can give you examples where PiComp will fail, there is no need to advertise them formally as something that a programmer should learn and memorize before embarking on a programming task. For one thing, it is hoped that all failures of PiComp to process a correct program will be detected and diagnosed (as opposed to producing incorrect but formally runnable code). Then, the compiler can be extended, if the discovered limitation proves serious enough.

The syntax of PicOS constructs

Here is a sample thread:

```
fsm receiver {
  address rpkt;
  entry RC_TRY:
      rpkt = tcv_rnp (RC_TRY, sfd);
  entry RC_SHOW:
      show (RC_SHOW, rpkt);
      tcv_endp (rpkt);
      proceed RC_TRY;
}
```

All threads in PicOS are declared with the same keyword *fsm*. Strands are differentiated from threads by an argument following the FSM's name, like in:

```
fsm sender (char *msg) {
  entry SN_SEND:
      address spkt;
      spkt = tcv_wnp (SN_SEND, sfd, plen (msg));
      spkt [0] = 0;
      strcpy ((char*)(spkt + 1), msg);
      tcv_endp (spkt);
      finish;
}
```

The argument must be a type name possibly followed by a variable name, which will override the default name **`data`**. Please read [picos] for details. The document contains an elaborate discussion of the new constructs and their semantics.

The most important difference w.r.t. the old (legacy) syntax that you should be immediately aware of is that local variables declared at the top of an FSM, e.g., **`rpkt`** in **`receiver`**, are now automatically rendered **`static`**. If you want to use truly local (transition-volatile) variables, you should declare them at states, like **`spkt`** in **`sender`**.

The PicOS document doesn't mention two new keywords, which are relevant only to VUEE: **`trueconst`** and **`idiosyncratic`**.

Normally, when you have a **`const`** variable, and you switch to a VUEE model, you want to have it "constant" on a per node basis (such that different nodes have different views

of that constant). This basically means that you want it appear as a node attribute. Consequently, PiComp will transform global **const** variables into node attributes (similar to other, non-**const** variables). In some situations, a **const** is really constant once for all, and corresponds to something that should be hardwired into the code flash of all nodes looking exactly the same everywhere (e.g., a fixed message string). In such a case, you can declare the constant as **trueconst**, as in this example:

```
static trueconst char ee_str [] = OPRE_APP_MENU_A
      "EE from %lu to %lu size %u\r\n";
```

While PicOS will just strip the "true" part of the keyword, VUEE will make sure to treat the declared item (or items) as actual constants (and place them in code rather than at the nodes). Note that **trueconst**, while being formally ignored by PicOS, may play an advisory role (as a useful comment) by referring to those items that are in fact supposed to be identical and immutable across all nodes ever running the program. For example, the program may use constants that are settable on a per-node basis, like a HOST ID being "burned" into the flash but with different values for different nodes. Such (mutable) constants should be declared as **const**, while true constants should be marked as **trueconst**.

In a multiprogram praxis [mkmk], the name of a **trueconst** variable is qualified to the program, i.e., the different programs can use the same name for their “private” true-constants.

One standard place where a **trueconst** specification is required (assuming the code has to compile for VUEE) is the declaration of a plugin handle [vneti], e.g.,

```
static int tcv_ope_null (int, int, va_list);
static int tcv_clo_null (int, int);
static int tcv_rcv_null (int, address, int, int*, tcvadp_t*);
static int tcv_frm_null (address, tcvadp_t*);
static int tcv_out_null (address);
static int tcv_xmt_null (address);

trueconst tcvplug_t plug_null =
       { tcv_ope_null, tcv_clo_null, tcv_rcv_null,
         tcv_frm_null, tcv_out_null, tcv_xmt_null,
         NULL, INFO_PLUG_NULL };
```

If the “true” prefix is absent, PiComp will have to set up **plug_null** as a node attribute, i.e., an attribute of a C++ class representing the node. But such an attribute cannot be initialized where it is formally declared: its initializer must be removed to a separate file where the **static** functions of the plugin will not be seen. So they cannot be static. If they are not, then they will become node methods (unless declared as **idiosyncratic**, see below). But then, pointers to them cannot be treated as pointers to functions, so they cannot be used as such in the initializer without a messy redefinition of **tcvplug_t** which would confuse the real-world compilation. This may sound complicated, but note that **plug_null** actually *should be* a (true) constant in the real-world program, the same and immutable for all nodes that run the program. So there is nothing inappropriate in declaring it as a “true”” constant.

Another VUEE-specific keyword is **idiosyncratic**, which can appear in front of a function definition or announcement, as in:

```
idiosyncratic int tr_offset (headerType*);
```


or

```
idiosyncratic int tr_offset (headerType *h) {
      return 0;
}
```

It is ignored by PicOS, but for VUEE it means that the function should be turned into a node method, rather than being defined as global or static. The difference becomes relevant, e.g., when you have a multiprogram praxis with different programs using the same names for some of their non-static functions. When such programs are linked together into a VUEE model, those names will cause linkage conflicts.

I have chosen the name **idiosyncratic**, because this prefix allows different functions with the same name, appearing in different programs of the same praxis (destined for different node types), to coexist within the VUEE model without upsetting the linker. In this sense, the keyword makes each of the possible multiple copies of the function *idiosyncratic* to one program of the (multiprogram) praxis.

When you call *picomp* with *-i* (see below), it will automatically implement all non-static functions as **idiosyncratic**, i.e., turning them into node methods. Do this whenever you have a multiprogram praxis with conflicting function names. With that option, the **idiosyncratic** keyword becomes superfluous. In fact, the *-i* option is rather popular for multiprogram praxes, because it is natural in such praxes to use similar functions (with identical names) in the different programs of the same praxis. When applying that option, you will occasionally need an exception from the blanket treatment of all (global) functions as idiosyncratic. Note that **static** functions are automatically exempt from that treatment: their names, being confined to a single file, cannot possibly collide in the linkage stage. If you need a (global) function to be non-idiosyncratic for some reason, put **nonidiosyncratic** in front of its declaration. This only makes sense with *-i*; otherwise the ugly specifier is ignored. One situation when you may want a function to be non-idiosyncratic is when it is passed as an argument to another function or its pointer is used, e.g., in an initializer (see above). This is because the signature of an idiosyncratic function (turned into a method) changes (compared to its original signature), so it will not fit the original type specification as the function argument or the (initialized) variable type.

Compilation

At the first level of approximation, you may assume that there is nothing special about a PicOS program being prepared for VUEE. As long as all the PicOS components referenced by the program have their VUEE models, it should be pretty much automatic. If you have to reference some features (symbols or constructs) associated with a specific board, sensor, etc., you may have to apply conditions, like:

```
#ifdef __SMURPH__
...
#else
...
#endif
```

However, you do not need them for structuring the program into separate blocks for PicOS and VUEE, nor you need to provide any VUEE-specific kludges, glues, headers, etc., as you did with the manual, legacy conversion.

Some syntactic caveats

Version 0.1 of PiComp (which nobody saw) couldn't handle compound variable initializers, e.g., ones like this:

```
struct MTag t = { 1, 2, 3 };
```

which had to be circumvented with ugly conditional compilation. The present version does handle such initializers. The problem with them is that when a variable becomes a node attribute (which happens to all non-automatic variables in a VUEE model), its initialization cannot be carried out statically, so one has to transform such declarative initializers into dynamically executed code. While there is no problem with a simple initializer, e.g.,

```
int sid = -1;
```

which can be trivially converted to an assignment statement, a compound initializer cannot be treated this way. PiComp handles such an initializer by declaring a dummy **static const** version of the variable (which can be initialized statically) and copying its memory area (**memcpy**) to the actual variable at initialization.

Note that, similar to plain C, you don't have to (dynamically) initialize those components of structures and arrays that are supposed to be zero. VUEE will first zero out the entire set of node attributes (the node's "global" RAM) and then do the detailed initialization.

One occasionally painful problem (only surfacing when compiling for VUEE) is that the names of local variables (of FSMs and functions) collide with the names of global variables, so they should be different. This is to say, you can use the same names for the local variables of different functions or FSMs, but when you name a global variable as one of the local variables, that will cause a conflict. The conflict will be detected at compilation time, usually with a cryptic message referring to the line of the local declaration. The problem is that global variables are replaced with macros looking like remote references (node_pointer -> node_attribute), and the appearance of the global name in a local declaration context will result in an attempt to declare something weird.

Another restriction that comes to my mind is that you cannot declare types (**struct**, **union**, **typedef**) with the same name in different files of the same program (it is OK in different programs of the same praxis). The reason is that wherever you declare your variables in the source program, they are eventually all put together as attributes of the same class in the VUEE model. Consequently, all the types used in the declarations of those variables are visible at the same time, and they must not collide.

PiComp may occasionally complain about something it cannot handle. Here is one example of a PiComp-generated error diagnostic:

```
app_peg.cc:27: picomp prohibits empty size in array typedefs
```

caused by a statement like this:

```
typedef arr_type [][3];
```

This illustrates a restriction on what kind of static/automatic data can be transformed into node attributes with definite memory layouts. Also, this kind of declaration:

```
int my_arr [][3];
```


will yield:

```
app_peg.cc:27: empty size only allowed with an initializer
```

As you can infer from the diagnostic, this:

```
int my_arr [][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

will work fine (PiComp will derive the size along the first dimension of **my_arr** from the initializer).

If you stumble upon something weird, please let me know and I will see what I can do. Note that despite the rather significant transformations undergone by the source program before it is submitted to the VUEE (SMURPH/SIDE) compiler, and subsequently to g++, PiComp tries very hard to tag the generated code with proper line sequencing marks, such that the errors reported by g++ should reasonably closely refer to the original source (which does not mean that they are never confusing).

Project structure

In terms of its layout as a collection of files, a praxis to be compiled by PiComp should essentially conform to the expectations of [mkmk]. The praxis may consist of multiple programs rooted at files named *app_xxx.cc*, where *xxx* is a program-specific tag (we shall call them *app* files). All files in the current directory (where the compiler has been called) whose names fit this pattern are assumed to represent separate programs of the praxis. if there is only one program, the app file can be named *app.cc*. Any remaining files contributing to those programs, including sources and/or headers, may appear within the same directory (at the same level as the app files) or in any of its subdirectories. Well, not quite *any*. Some subdirectories are excluded based on their names which, among other things, allows you to keep other stuff in the praxis directory without having to worry that it might be accidentally “compiled”. One exact name excluded from search is *VUEE_TMP*, i.e., the directory where PiComp puts the intermediate files for SMURPH compilation. Additionally any directory whose name (ignoring the case) starts with *KTMP* or contains any of the strings *JUNK*, *ATTIC*, *OSSI* is ignored as well, regardless of its level.

Headers are sought automatically as needed and can be placed into any (qualifying) subdirectories, not necessarily ones related to those containing the referencing files. A source file must be referenced either from the app file or from one of the other files already deemed to belong to the set (be it a header or a source) via the special comment sequence //+++ (as described in [mkmk]). For example:

```
//+++ app_diag_peg.cc "lib_app_peg.cc" <msg_io_peg.cc>
```

adds three source files to the program's set (the delimiters are optional). All the subdirectories of the current directory (with the exceptions noted above) are scanned automatically for files requested this way (as well as for any **#included** headers). There is no need to specify exact paths. For example, the file *lib_app_peg.cc* referenced above can be provided in any subdirectory (at any level) of the praxis's main directory.

Call arguments

When you invoke the compiler as:

```
picomp
```

it will do the following things:

1. Check for the completeness of the environment (in particular VUEE and SIDE).

2. Scan the current directory for app files and build the list of tags representing the programs belonging to the praxis. If there is an app file named *app.cc* (empty tag, a single-program praxis), then no other app files are legal (the compiler will complain if it sees any).

3. For each program, the compiler will parse all source files belonging to the program and create their compiled versions in directory *VUEE_TMP*. Any headers referenced by those files will be expanded in (so they will not show up in *VUEE_TMP*). The compiler will add two extra files for each program: the VUEE header (included by all compiled files) and the VUEE initializer (containing code for starting and resetting the node model),

4. Having processed all programs, the compiler will add one more file to *VUEE_TMP* containing the root process of the SIDE simulator and providing a glue for the (semi-independent) models of different nodes (for the multiple programs).

5. Finally, if there were no errors thus far, the compiler will invoke the VUEE compiler (*vuee*) on the produced set of files. If the compilation succeeds, the executable (*side* or *side.exe*) will be moved to the praxis main directory.

You can say:

```
picomp -n
```

to omit the last step of the above sequence. Note that you can then move (*cd*) to *VUEE_TMP* and execute *vuee* by hand.

If you say:

```
picomp -- ...
```

then any arguments following `--` will be passed to *vuee* (rather than being interpreted by *picomp*).

When invoked this way:

```
picomp -p
```

the compiler acts as a filter (stdin to stdout) to translate a PicOS source file into C (for a real-world compilation). You are not supposed to ever have to do that by hand. Such calls are inserted by *mkmk* into *Makefiles* at the pertinent stages.

Here are the remaining parameters:

`-v`	Selects the VUEE compiler (as opposed to the PicOS compiler). This is the default (i.e., the counterpart of **`-p`**)
`-i`	Forces all functions to be compiled as idiosyncratic, except for those that are explicitly declared as **`static`** or **`nonidiosyncratic`**.
`-e`	Invokes the SMURPH compiler on the existing contents of *VUEE_TMP*, e.g., created previously with **`picomp -n`**.

-3	Selects the 3d option for VUE^2 compilation, i.e., the network nodes will be deployed in 3-space as opposed to a 2d plane.
-G [fn]	Produces debug output. The optional extra argument (recognized if the following argument does not start with a minus sign) is the name of the file where the debug information is to be written. If not specified, it defaults to standard output (in the VUEE mode) or standard error (in the PicOS mode). Note that PiComp acts as a filter (writing its output to stdout) in the PicOS mode.
-A	Stops on the first compilation error (diagnosed by PiComp).
-D	Defines a symbol (as explained below).
-H	Defines an extra header file (as explained below).
-X	Defines an exception for a multi-program praxis (as explained below)
-V	Just prints the version number and exit.

When you specify **-n** together with **-e**, the compiler will do nothing, but only list the status of the present contents of *VUEE_TMP* (for example “OK, no errors, no warnings” means that the last compilation fully succeeded, so **-e** should now work).

Arguments **-D** , **-H**, and **-X** are illegal with **-p**, i.e., they only apply to VUEE compilation. You can use **-D** to define symbols for the compiled model. The letter **D** should be immediately followed (no space) by one of these constructs:

symbol
symbol=something
tag+symbol
tag+symbol=something

The last two variants will set the specified symbol only for the specified program (of a multiprogram praxis). For example:

```
picomp -Dpeg+BOARD_TYPE=WARSAW
```

will set **BOARD_TYPE** to **WARSAW** for the *peg* program. Multiple arguments of this kind are allowed, e.g.,

```
picomp -i -Dpeg+BOARD_TYPE=WARSAW -Dtag+BOARD_TYPE=ARTURO_PMTH
```

In a similar way, **-H** (which also can occur multiple times) declares an extra header file to be implicitly included at the very front of a praxis source file, logically following the definitions inserted with **-D**, but preceding everything else. The letter **H** should be followed by a file name optionally preceded by ***tag+***, e.g.,

```
picomp -Hsomefile.h
picomp -Hpeg+../../PicOS/MSP430/BOARDS/WAR/board_options.sys
```

In the last case, the extra header will only be prepended for the specified program (of a multiprogram praxis).

Arguments of these types are used by PIP to pass to the program the board information [pip]. This way the program(s) of a VUEE model compiled under PIP automatically receive board information (symbols **BOARD_TYPE** and also **SYSVER_B** –

see [mkmk]) in the same way as for a program compiled for a real board. They can also see symbols defined in options files associated with the respective boards.

If the praxis consists of multiple programs, then, by default, all those programs are compiled into the VUEE model. With **-X**, it is possible to declare an exception, i.e., a program that should not be compiled into the model. The suffix of that program must immediately follow the letter **X**. Similar to **-D** and **-H**, **-X** can occur multiple times, so it is possible to specify more than one exception. For example:

```
picomp -Xa319t -Xwarp
```

will ignore (not compile) programs rooted in files **app_a319t.cc** and **app_warp.cc** in the current praxis's directory.

Parameterization issues for multi-program praxes

One tricky issue related to multiprogram praxes (this applies to VUEE compilation only) is their parameterization with extra header files. Those headers are referred to as *options* files and they may be coming from several places:

1. For a single-program praxis, the basic header file is automatically sought as *options.sys* in the project's directory (i.e., the directory containing the *app.cc* file). This file is not mandatory.

2. For a multiprogram praxis, the basic header file is program-specific and named *options_xxx.sys*, where xxx is the program file suffix (corresponding to *app_xxx.cc*). If a file with this name doesn't exist, PiComp will try *options.sys*. Again, it is OK if both files are absent. Note that a program-specific options file overrides the single *options.sys* file which, in principle, can be shared by all programs of the praxis.

3. PiComp can be instructed to use other header files (see the -H option in the preceding section), typically by PIP. Those files are are combined with the options files located in the project's directory by being put in front of those files. In this sense they take precedence, but in contrast to, say, options_xxx.sys versus options.sys, they do not replace (override) them.

Any symbols defined in the headers described above are available to the program to which they pertain. A subset of those symbols (selected from all the header files for all the programs) is used to parameterize the shared components of the model. This is where the tricky part comes in. It stems from the fact that some components of the model must be parameterized in exactly the same way for all programs, even if those programs happen to have different ideas. The issue may be ameliorated in the future. For now, it concerns TCV (VNETI), and the radio driver. Components like the UART model (which is parameterized in the data file on a per-node basis) are generally immune.

The predefined set of symbols mentioned above is known to PiComp (the description is kept in file *modsyms.h* in the Picos directory). Some of those symbols, if defined in any of the extra header files, must be defined in the same way in all of them. Some other symbols can be redefined in different header files (causing no serious problems), but then PiComp will issue a warning and assume a single definition that is considered safe for all cases.

For illustration, **TCV_HOOKS** defines a VNETI option that affects the layout of buffer headers. As there is a single implementation of VNETI in the model that must be shared

by all nodes, the option cannot be defined differently in different programs, because the VNETI headers (and VNETI data structures) used by (some of) them would be inconsistent with the actual data structures used by VNETI. Thus, PiComp will signal a fatal error when an attempt is made to define the symbol differently in different per-program extra headers.[2]

A bit about the internals

I am only discussing here the PicOS-to-VUEE translation. This is all very preliminary. I may write more later. Things may still change without a warning.

The compilation is heuristic, but it should work for all (or most of) sane cases. I am almost sure that the compiler will never produce incorrect code without a warning. Other than that, it may fail on good and healthy source. Whenever that happens, I will do my best to fix the problem, if it can be fixed at all.

This is how it works in a nutshell:

Having determined the list of programs to compile (note that one program may consist of multiple files), PiComp proceeds to handle them one at a time. No information survives from one program to another (note that "program" is not synonymous here with "file"), except for the global list of tags (the *xxx* parts from *app_xxx.cc*), which are needed at the end to generate the root file of the SMURPH simulator (also called the glue).

When processing a program, the compiler starts from its app file. The file is read and scanned for preprocessor directives (conditions and includes). All files **`#included`** by it are read as needed and processed recursively. An included file that belongs to VUEE (comes from *VUEE/PICOS*), as opposed to the praxis, is not fully expanded (it will be included later again for the "true" compilation), but its definitions are preserved (and conditions are obeyed), such that the resultant output file from that stage is devoid of macro references and conditional code. Note that macros must be pre-expanded (before parsing) as they may produce (translate into) PicOS keywords or type declarations.

The stage produces a set of preprocessed (expanded) files comprising the program. Note that all those files are source files (all praxis-specific headers have been already incorporated into them). They may still reference things (but, notably, not macros) that are not defined within them, but all such things share one property: they already belong to the virtual world (i.e., to the model), so PiComp doesn't have to worry about transforming them into anything different from what they are.

Each of those files is subsequently parsed and processed. The processing order is exactly the same as that of their addition to the set (as requested with the special comments //+++), with the app file going first. This is what happens:

All type declarations (**`typedef`**, **`struct`**, **`union`**), including those implicit in variable declarations, are identified and removed. They are stored in a list (eliminating duplicates and detecting any conflicts) to be included as a single set of type declarations for the entire program (to be provided in a special header file). The reason is that the (necessarily single and centralized) declaration of the node class, including all its attributes, must in principle be able to see all those data types.

Function announcements are left intact except for those announcing **`idiosyncratic`** functions, which are turned into macros transforming references to them into references

[2] TARP used to suffer from this deficiency, but not any more. There is a clear path to making VNETI immune to it as well.

to the corresponding node methods. Similarly, actual definitions of **idiosyncratic** functions are transformed into mangled definitions of node methods. The signatures of those methods are stored for inclusion (declaration) within the node class.

Global variable declarations are analyzed and split into lists of variables, including their types and initializers. Those lists are stored for inclusion (declaration) as node attributes. The original declarations are removed and replaced with macros referencing those (mangled) attributes via the global node pointer. Static declarations are treated the same way, except that their mangling is file-specific. This way, the same static name appearing in two different files will translate into two different attributes. This closely resembles what you were doing manually, except that now, with all this being done by a program, we can afford to be more exacting and careful.

FSM's are transformed into code methods of SMURPH processes. Information regarding their names, types (thread/strand), and lists of states is stored until all files have been handled. Also, *local* declarations within FSM's (which formally become static) are transformed into macros referencing the appropriately mangled node attributes. Those macros are **#defined** in front of the code method and **#undef**(ined) at its end. This way, the “local” variables (turned static) are only visible within the FSM. Note that there is no need to do anything about the automatic declarations (appearing at states). FSM states are collected, to be inserted into the definition of the corresponding SMURPH process class.

PicOS-specific keywords, like **runfsm**, **proceed**, are identified and processed as required.

When all files of the program have been processed, PiComp generates a header file to be included by all of them. That file combines all the declarations of types extracted from all the program files and follows them by the declaration of the node type along with all the attributes and methods. Then it declares the process types for all the FSM's.

Finally, one more source file is created which contains the reset code to initialize the node attributes and to start the root FSM. Any dummy variables needed by compound initializers are declared as **static const** within that file.

Having handled all the programs, PiComp creates the glue file containing the root process of the simulator. Then, if there were no errors (and the compiler wasn't called with -n), PiComp invokes *vuee* to compile the contents of *VUEE_TMP*.