mkmk

the PicOS Makefile Maker

June 2012

1 Preamble

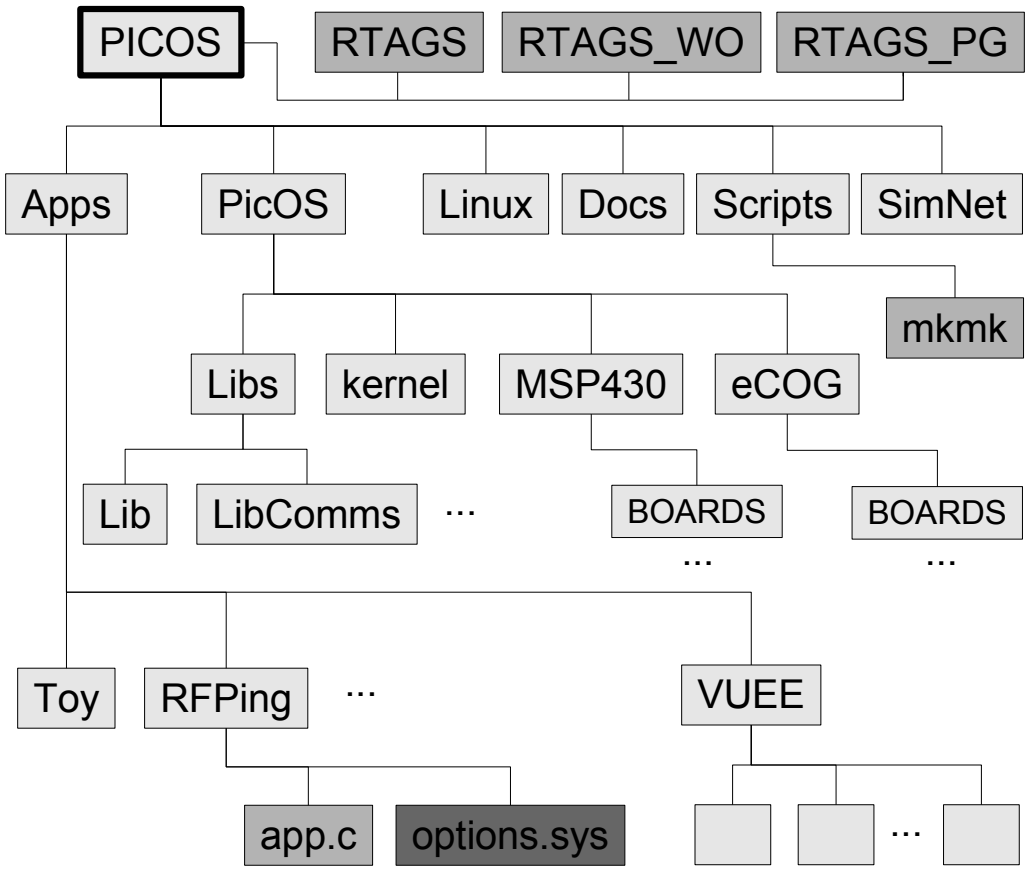
This document describes the **mkmk** script whose role is to create Makefiles and/or projects for compiling PicOS praxes into loadable images. The script originated as a crude and simple hack, but, over the years of its incessant tweaking, has evolved into something complicated enough to deserve a document outlining its operation.

2 The general idea

The script is typically executed in a directory containing the source code of a PicOS praxis. Its goal is to determine the complete set of files to be compiled into the target loadable image. It begins by reading the first (anchor) source file (typically named **app.c** or **app.cc**), which usually contains **`#include`** statements, which in turn force the script to read other files, and so on. Based on some hints that may be present in the processed files, the script may add to the pool some source files that are not explicitly included. At the end of its operation, **mkmk** may produce a Makefile (to be used as input for **make**) or a CyanIDE project.

3 The principle of operation

The PicOS package comprises a hierarchy of files: kernel source files, device drivers, libraries, documentation, and so on. Application programs (called praxes) are also part of that hierarchy. Its skeletal and partial layout looks like this:



A praxis to be handled by **mkmk** must be located somewhere under **Apps** (it doesn't matter how deep). The script looks up **Apps** going up from the current directory and then, using that directory as a reference point, determines the positions of all the other relevant components of the package.

The simplest and most popular usage of **mkmk** is to create a Makefile for a praxis contained in some directory under **Apps**. Note that the script *must* be called in that (target) directory. The directory contains at least one source file of the praxis named **app.cc**.[1]

The script will find out which other files must be compiled and linked together with **app.cc** to make the program complete. Some of those files may be provided by the user (and reside in the same directory as **app.cc** or, perhaps, in its subdirectories), others will belong to the system (i.e., the kernel or some standard library). In any case, the generated Makefile will prescribe the full compilation of all the requisite source files, i.e., the scheme assumes no precompiled object libraries, except for the standard (non PicOS) libraries that come with the C compiler.[2] This is because our philosophy of parametrizing the system favors a recompilation of its modules for the different sets of options. This is OK because the system is small.[3] Besides, once you run the Makefile generated by the script through **make**, you will effectively obtain an object library (a set of object modules) tailored to the specific praxis with its specific set of options. Then, whenever some fragment of the praxis is modified, only the modified files and their dependents will be (automatically) recompiled when you run **make** again.

Another source of input to the script (in addition to **app.cc**) is the board description, which is contained in a special directory. A praxis is always compiled for a specific board described by a set of files that are **`#included`** by some system files and also, implicitly, by the praxis program (which always **`#includes`** at least **sysio.h**). Intentionally, those files refer to a specific model of the CPU with a specific set of devices, pin connections, and so on. Supposing that the root directory of the PicOS package is named **PICOS**, the board descriptions are kept in subdirectories located at **PICOS/PicOS/*/BOARDS**, where the single path component represented by the asterisk corresponds to the architecture. At present, it is either **MSP430** or **eCOG**.

Note that the directory **PICOS/PicOS** contains subdirectories other than **MSP430** and **eCOG**. The script is preconfigured to look for **BOARDS** (and its subdirectories containing board definitions) only in **MSP430** and **eCOG**. The minimum required set of arguments to the script includes a single argument viewed as the name of the directory containing the board description. For example, when you run this:

```
mkmk CHRONOS
```

the script will scan the directories **PICOS/PicOS/MSP430/BOARDS** and **PICOS/PicOS/eCOG/BOARDS** (in this order) for a subdirectory named **CHRONOS** and, if such a subdirectory is found, it will use its contents as the requisite board definition files. Note that it makes little sense to use the same board names for different architectures.

[1] Or, possibly, **app.c**. With the "new" programming constructs added to PicOS (and the introduction of PiComp – as of version R100605B), the extension **.c** (as opposed to **.cc**) applies to legacy code (which is not preprocessed by PiComp). While the present version of **mkmk** does handle legacy code, that case receives only a marginal treatment in this document.

[2] A precompiled library can be used in the library mode described later.

[3] The primary culprit responsible for this approach is CyanIDE (the SDK for eCog-based boards by Cyan Technologies), whose libraries were extremely clumsy and basically useless as an approximation of object libraries indexed by entry points. Now, as we are departing from the CyanIDE-inspired model (retiring eCog and CyanIDE support), we may (and probably should) reconsider the model.

The script assumes very little about the interpretation of the particular files contributing to the target project. Basically, their relationship is determined by the configuration of **`#include`** statements of the C preprocessor in the files parsed by the script. The "standardized" layout of board description directories is a matter of convention implied by the layout of PicOS sources and their expectations in terms of some "external" header files that they want to **`#include`**.

Depending on whether the specified board description directory is found under **MSP430** or **eCOG**, the script assumes that the rest of its activities will be oriented towards the specific CPU and, more importantly, its specific SDK. For MSP430, the script can only generate Makefiles for mspgcc; for eCOG, it can generate Makefiles as well as projects to be interpreted by CyanIDE, which is Cyan Technologies' SDK for eCOG-based platforms.[4] We shall describe in detail what happens for MSP430 (building a Makefile for mspgcc). Building a Makefile for eCOG (CyanIDE) is similar, so we will be discussing the differences along the way. Then, we will elaborate a bit on the eCOG project case.

3.1 The CPU model

In the first step, the script opens and reads the file **params.sys** in the board description directory. This file is expected to contain lines in the form of:

```
-x parameters
```

where ***x*** is a letter and ***parameters*** is some parameter string. Two parameters are implemented at present: **`-c`** provides the CPU model and **`-l`** gives the list of files to be compiled into a library for a special type of **mkmk** invocation intended to create an object library for the board (as described further in this document). The **`-l`** parameter is optional and only needed if you ever want to build a library for the board. If the first non-blank character of a line in **params.sys** is not **`-`**, it is assumed that the line is preceded by **`-c`**, i.e., it provides the CPU model (so the **`-c`** prefix is optional).

For MSP430, there is a multitude of CPU models with different hardware properties that must be identified to the C (gcc) compiler and (in many cases) also to PicOS. The role of **`-c`** in **params.sys** is to identify the model that should be assumed for the board. The actual parameter is a string looking like *name***x***number,* e.g., **msp430x1611** or **cc430x6137**. The default CPU model, assumed when the **params.sys** file is absent or it doesn't provide the model, is **msp430x148**.

For eCOG, the **`-c`** parameter is ignored.

3.2 Standard #defines

The C compiler that will be invoked by the Makefile built by the script will receive some standard symbol definitions that can be used by the praxis (and kernel) source code for parameterization. Here is the list of those definitions:

```
#define SYSVER_U  major version number
#define SYSVER_L  minor version number
#define SYSVER_T  GIT RTag
```

[4] Only praxes in legacy format (i.e., ones not using the PicOS-specific programming constructs, like **fsm**, **runfsm**, etc.) can be turned into CyanIDE projects. This is because CyanIDE projects cannot preprocess source files via PiComp. While, theoretically, we could try to do something about it, the relevance of CyanIDE SDK for the future of PicOS is at best questionable at this time.

These symbols identify the system version. As of the time of writing, the PicOS version is 3.5, so **SYSVER_U** will be defined as 3 and **SYSVER_L** as 5. ***GIT RTag*** is a string of the form **ppyymmdd*M*** (where ***pp*** consists of one or two letters and ***yymmdd*** is the date and ***M*** is a letter - typically **A**) identifying the current revision, e.g., **PG120620A**.

```
#define __MSP430__
```

or

```
#define __ECOG__
```

depending on the target CPU architecture. Only one of the two symbols can be defined at a time.

```
#define BOARD_xxxxx
#define BOARD_TYPE      xxxxx
```

The two constants allow the praxis to know the board for which it is being compiled. The string **xxxxx** in both cases stands for the name of the board definition directory, e.g.,

```
#define BOARD_CHRONOS
#define BOARD_TYPE      CHRONOS
```

Here are two more symbols that allow the program to know its label (see Section 5):

```
#define PGMLABEL_xxx
#define PGMLABEL  xxx
```

where ***xxx*** is the label of the currently compiled program. These symbols are only set if:

1. the praxis actually consists of multiple (labeled) programs

   and

2. the praxis is compiled in multiple-Makefile mode (Section 5.1), i.e., the multiple programs are “made” independently

The above list only covers those symbols that are introduced explicitly by **mkmk**. PicOS files **#included** by the praxis (like **sysio.h**) provide other definitions that are derived from the above symbols or from some other symbols defined by the C compiler itself. For example, gcc defines a symbol looking like **__MSP430_*xxxx*__** or **__CC430_*xxxx*__** identifying the CPU model. This definition is derived from the CPU model parameter extracted from **params.sys** (**-c**) and passed to the compiler in the Makefile. Also, **sysio.h** defines additional version-related symbols being various transformations (e.g. stringifications) of the three basic system version symbols inserted by **mkmk**.

For an example, note that **BOARD_TYPE** is set to a symbol which is (usually) not defined. You may prefer to have that identifier in a textual form, e.g., to include it in some (output) string. One way to “stringify” such a symbol is to take advantage of the **stringify** macro, e.g.,

```
#define BOARD_NAME stringify (BOARD_TYPE)
```

Note that this trick also applies to **PGMLABEL**. If **BOARD_TYPE** is defined at all, **sysio.h** will automatically create a stringified version of the board type as:

```
#define SYSVER_B  stringify (BOARD_TYPE)
```

3.3 The role of RTAGS

The root directory of **PICOS** contains a number of files whose names begin with **RTAGS**. Those files contain the revision history, including revision tags, as well as the system version numbers. Originally, there was a single **RTAGS** file, but since revision PG100628A we introduced separate files, named **RTAGS_xx**, to be used by different developers (**xx** being their initials).

The script obtains the most recent version number and revision tag (to be assigned to **SYSVER_U**, **SYSVER_L**, **SYSVER_T**) by scanning all **RTAGS** files. A revision tag is recognized as a string starting with one or two letters at the beginning of a line and followed by exactly 6 digits, followed in turn by a letter and a colon. For each **RTAGS** file, the script selects the last tag (lexically) and from that set it picks the one with the largest 6-digit value. This is the tag that will be returned in **SYSVER_T**.

Then, **mkmk** looks at the lines that begin with tags and contain two numbers following the colon and separated by a sequence of white spaces, e.g.,

```
PG100823:   3     2
```

The line of this form with the most recent tag value (the largest 6-digit number) is chosen from all **RTAGS** files, and the two numbers following the tag are returned as **SYSVER_U** and **SYSVER_L**, respectively.

3.4 Collecting the files for compilation

Next, the script constructs the set of all files contributing to the target built. The set starts with a single element: the file **app.cc** which must be present the current directory, i.e., the directory in which **mkmk** has been invoked. The script parses that file effectively mimicking the behavior of the C preprocessor. Any file **#included** by **app.cc** (which is not already in the set) is added to the set and marked for parsing. This procedure continues recursively until there are no more files to be added to the set.

While searching for **#included** files, **mkmk** first looks into the directory containing the referencing file and then traverses the system directories, followed by the praxis directory and its subdirectories. The order is:

```
"PicOS/architecture"
"PicOS/architecture/BOARDS/target_board/"
"PicOS/kernel"
"PicOS"
"PicOS/Libs/*"
"."
"./*"
```

In the above list, ***architecture*** stands for **MSP430** or **eCOG**, and ***target_board*** is the name of the board definition directory. The praxis directory is denoted by **.**, and **/*** means that all subdirectories of a given directory (recursively, to an arbitrary depth) are also being searched. For those subdirectories, the search order is alphabetic with respect to their global path names.

Notably, some subdirectories of the praxis directory are excluded from the search based on their names. While this feature may appear a bit clumsy at first sight, it is needed, it is

effective, and it causes no problems. Two (exact) directory names always excluded from search are *CVS* and *VUEE_TMP*. The latter is the directory where PiComp puts the intermediate files for VUEE 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. The exclusion principle applies recursively to all subdirectories of the praxis directory regardless of the level at which they occur.

If multiple versions of a file with the same name occur at different places of the tree, it may be difficult to guess which one will be used. The script issues a warning in such a case. If you really want to have the same (tail) name for multiple files, you can prefix the file name in the **`#include`** statement with a partial path uniquely identifying one specific file, e.g.,

```
#include "Libs/LibComms/net.h"
```

The prefix can be any trailing component of the complete path to the file, but the file must be actually located in one of the officially searchable directories. For a praxis-specific file, this means the praxis directory or any of its subdirectories (at any level).

In any case, **mkmk** will warn you if your specification is still ambiguous, i.e., the partial path still describes more than one file, so you can always fix the problem by providing more path components.

It is possible to indicate that a particular source file should be added to the pool of files to compile, even though that file is not explicitly **`#included`**. This is necessary as, usually, only **`*.h`** files are explicitly included, while some **`*.c`** files may be needed to provide the actual code behind the headers. This is accomplished by using a special C comment looking like this:

```
//+++ "filename" "filename" ...
```

There should be no space between **`//`** and the first **`+`**. The file names must refer to source files (i.e., the only legitimate extensions are **`.cc`**, **`.c`**, and **`.asm`** (the latter only applicable to CyanIDE.

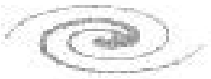
Similar to **`#included`** files, path prefixes can be used to discern different files with the same tail names. Also in this case, **mkmk** will warn you when the specification is ambiguous.[5]

3.5 Conditional processing

In addition to interpreting **`#include`** statements appearing in the parsed files, the script mimics the behavior of the standard C preprocessor in its attempts to determine which portions of the parsed files are in fact active. This means that it tries to evaluate **`#if`** and **`#ifdef`** statements. This is somewhat heuristic[6] but reasonably complete, although **mkmk** may not be able to authoritatively evaluate some legitimate expressions. For one thing, the script doesn't know the values of all symbols as, for example, it doesn't parse the library headers that come with the compiler. If **mkmk** cannot evaluate a condition, it will issue a warning and assume that the condition is false. Note that an **`#ifdef`** or **`#ifndef`** on a symbol not known to the script, which becomes known when the file is in fact compiled, may tacitly produce the wrong outcome.

[5] This doesn't really work for **`.c`** /**`.cc`** files, i.e., files with the same tail names will always result in a conflict as they will be ultimately mapped into an object (**`.o`**) file with the same name stored in **KTMP**. I may do something about this in the future, if it causes problems.

[6] The evaluation algorithm was greatly improved with revision R100509A.



Errors in the evaluation of conditions by **mkmk** are only meaningful, if the erroneously skipped or included piece contains **`#includes`** (or **//+++** constructs) that should have been interpreted/skipped. If you know that a particular condition is harmless from that point of view, and you want to eliminate the warnings produced by the script, you can insert this statement:

```
#define mkmk_eval v
```

in front of the respective condition. For as long as **`mkmk_eval`** remains defined, the script will ignore all conditions effectively assuming that they evaluate to ***v***, which should be 0 or 1 (if missing, it is assumed to be zero). You can revoke this setting by undefining the symbol, i.e.,

```
#undef mkmk_eval
```

3.6 The Makefile

At the end of an error-less Makefile-type run, the script will produce a Makefile (named **Makefile**) in the praxis directory, i.e., in the directory in which **mkmk** was executed. The contents of the Makefile are similar for MSP430 (mspgcc) and eCOG (CyanIDE), but, of course, different programs (compilers, linkers, etc.) are invoked to handle the respective files. Also, in the case of eCOG, some extra files must be copied to the praxis directory, owing to the exotic requirements of CyanIDE.

For MSP430, along with the Makefile, **mkmk** will create a directory named **KTMP** and two files: **gdb.ini** and **.gdbinit** with identical contents. The files provide startup data for **msp430-gdb** (the GNU debugger that comes with mspgcc), such that it can be used with the GDB proxy (**msp430-gdbproxy**) over the JTAG interface. This is covered in a separate document that comes with mspgcc. The two version of the initialization file are the artifact of the different versions of **msp430-gdb**, which expect different names of that file.

For every praxis source file whose extension is **.cc** (as opposed to the legacy extension **.c**), the Makefile will include an invocation of PiComp to preprocess the file. The output from that preprocessing is stored in **KTMP** from where it will be compiled by the C compiler. Legacy praxes need no preprocessing of this kind: their source files are submitted directly to the C compiler.

When you execute **make** in the praxis directory, the source files collected by the script will be compiled and their object (**.o**) versions will be stored in **KTMP**. Then, those files will be linked into an ELF executable called **Image**. This file can be loaded into the target board, e.g., with **msp430-gdb** (via **msp430-gdbproxy**). An Intel-hex version of the image file, named **Image.a43**, is also provided. This file can be handled with other loaders, e.g., the Elprotronic FET programmer.

For CyanIDE, instead of **KTMP**, the script creates two directories: **out** to contain the output files from the compilation (CyanIDE generates a whole bunch of them) and **temp** to store the **.asm** files (which can be viewed as CyanIDE equivalents of object files) as well as the output file from PiComp (for those source files whose names end with **.cc**). The **.lif** files, used by the CyanIDE linker, are stored directly in the praxis directory.

Whenever you invoke **mkmk** in a given praxis directory, the script starts by removing from it all files and directories that it thinks are the results of a previous **mkmk** or **make** run. It doesn't discriminate between mspgcc and CyanIDE, so, in particular, it will remove the directories named **KTMP**, **out**, and **temp**, as well as all files whose names

end with **.lif**, **.asm**, or **.cyp** (for CyanIDE projects – see below) plus the two initializers for **msp430-gdb**. Make sure to never use those file names for anything non-volatile.

3.7 CyanIDE projects[7]

When you invoke the script this way:

```
mkmk boardname -p
```

it will create a CyanIDE project. The root file of the project will be called ***xxxxxx*.cyp**, where ***xxxxxx*** is the tail name of the praxis directory. This will only work if the specified board name belongs to the **eCOG** part of the tree (the board definition is present in **PicOS/eCOG/BOARDS**). By clicking on that file, you will open the project and get into the interactive SDK of CyanIDE.

3.8 Comments on options.sys

When the PicOS project started, there was no clear concept of a board definition. The praxis directory was then supposed to contain a file named **options.sys** providing definitions of some constants and macros determining system parameters, such as the configuration of device drivers, crystal options, and so on. After introducing explicit board definitions (with revision R061030A), the role of **options.sys** became secondary (part of its original role was taken over by **board_options.sys** in the board definition directory) and questionable, although, until R100217A, that file was still required. Its primary use was to (temporarily) override some of the official board options and, perhaps more importantly, to provide declarations for VUE[2] selecting the optional library components. Starting with R100217A, **options.sys** is optional. Its presence or absence is detected by **mkmk** and its inclusion (by **sysio.h**) is conditional upon its presence.

4 Library modes

Sometimes you may want to create a customized PicOS environment for a specific board by precompiling all the requisite system files (e.g., board-specific device drivers) into an object library. For example, you can give the developers the object library without the need of providing them with the system source files.

When called this way:

```
mkmk boardname -l
```

the script will create a Makefile for building an object library for the specified board. While the directory where the above invocation is made must be a subdirectory of **Apps** (possibly at a nontrivial depth), the directory need not contain **app.cc** and, if it does, that file (as well as any other files present there, except for the optional **options.sys**) will be ignored.

When called this way, the script expects to find in **params.sys** in the board definition directory the parameter `-l` specifying the (space-separated) list of system files to be included in the library. These can be both source files (with names ending with **.c**) as well as header files (with names ending with **.h**). Generally, you don't have to specify all files that you want to add to the library, as the script will automatically determine dependencies (in the way described earlier) and automatically add any files included (or requested) from the ones already incorporated into the set.

[7] CyanIDE projects only work for legacy praxes.

If the directory in which **mkmk** is invoked contains **options.sys**, that file is included the standard way after **board_options.sys** from the board definition directory.

At the and of a successful run, the script will generate a Makefile. By running **make**, you will create a directory named **LIBRARY** in the current directory. In the case of MSP430 (mspgcc), that directory will consist of the following items:

- Subdirectory **include** containing the header files. These are all the header files (plus all their dependent headers) **`#included`** from the source files compiled into the library, as well as the header files explicitly specified with the **`-l`** parameter in **params.sys** in the board definition directory (plus any files **`#included`** from them).

- File **libpicos.a** constituting the object library of the selected modules and their dependent modules (e.g., requested with **//+++**).

- File **params.sys** being the exact copy of **params.sys** from the board definition directory.

- File **target** consisting of a single line identifying the CPU model and architecture.

For CyanIDE, the **LIBRARY** directory contains the following elements:

- Subdirectory **include**, playing the same role as in the mspgcc version.

- Subdirectory **asm** containing **.asm** files of the compiled modules. This directory can be viewed as an equivalent of **libpicos.a** from the mspgcc version.

- Files **params.sys** and **target**, as for the mspgcc version.

- Subdirectory **stubs** occasionally needed to contain some trivial source files that cannot be precompiled under CyanIDE.

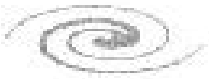
Once a library has been created, it can be referenced, instead of the board definition directory, to build executable praxes. For example, when called this way:

`mkmk -M` ***`library_directory_path`***

the script will create a Makefile to build the praxis (located in the current directory) using the specified library of precompiled modules. Similar to a standard call (specifying a board directory), the script expects to find **app.cc** in the current directory. The resulting Makefile prescribes how to build the praxis by referencing the header files in the library's include subdirectory and its precompiled modules. The praxis can only reference those files/modules incorporated into the library, as well as any files present in its own directory and subdirectories.

The board name assumed with a library-based build (affecting the definitions of symbols mentioned on page 5) is equal to the tail name of the library directory. Thus it may make sense to name the libraries as the boards for which they have been created.

In principle, a library directory can be moved anywhere and renamed to anything. However, any invocation of **mkmk** creating a Makefile for building a praxis (inlcuding a library-based build) must be made from a subdirectory of **Apps**. The script also needs



the **RTAGS** files (from the root directory of the package) which it expects to find at the same level as **Apps**. In other words, a rudimentary layout of the complete PicOS package must be present also for a library-based build. This layout must consist at least of subdirectory **Apps** and one file whose name starts with **RTAGS**, with the library (or libraries) located anywhere (e.g., at the same level as **Apps**). The RTAGS file is needed to determine the system version and its revision tag.

For CyanIDE, it is possible to create a library-based project[8] by specifying **`-P`** (instead of **`-M`**) as the first argument to **mkmk**. This will only work if the library specified as the second argument has been created for CyanIDE. All the remaining rules are the same as for a library-based Makefile build.

5 Multiprogram praxes and VUE^2 hooks

The term "praxis" refers to a complete system, which, in general, may involve multiple programs. So far we have assumed that the praxis directory contains a single program, with our objective defined as turning that program into a (single) executable file that can be loaded into a board. However, a praxis may consists of several different programs that may be loaded into boards of different types (e.g., data collectors and access points). While it may make sense to keep such programs isolated from each other (in separate directories) and treat them as separate projects, storing and viewing them together has some advantages. The most important advantage is the possibility to make all those programs available to **VUE^2** as a single executable model in SMURPH/SIDE.

5.1 Single Makefile for multiple programs

According to the preceding sections, when generating a praxis Makefile, **mkmk** seeks a file named **app.cc**, which it treats as the “anchor” file of the praxis program. The configuration of its include files + the board options + the possible extra options specified in **options.sys** in the praxis directory determine the constituents of the target Image file to be loaded into the microcontroller. The full picture is in fact a bit more complicated, with the intention of providing reasonably flexible hooks for handling praxes consisting of multiple programs.

If the praxis directory contains no file named **app.cc**, but there is another file whose name looks like **app_*xxx*.cc**, where ***xxx*** (called the program's *label*) is any sequence of letters and/or digits, **mkmk** assumes that this file is the root source of the praxis and proceeds from there. The resulting image files built by the produced Makefile will be called **Image_*xxx*** (the ELF version) and **Image_*xxx*.a43** (the Intel hex version). If multiple files named **app_*xxx*.cc** are present, e.g., **app_one.cc**, **app_two.cc**, and **app_three.cc**, **mkmk** will assume that there are multiple programs in the directory (three in this case) and generate a Makefile to compile them all into separate images (**Image_one**, **Image_two**, and **Image_three**). A VUE^2-compliant multiprogram praxis will follow this convention to make the multiple programs discernible by **mkmk** (for PicOS), while keeping them unified for the purpose of having a single global model of the praxis in VUE^2.

When handling a program anchored at a labeled **app** file, the script changes a bit its procedure of locating **options.sys**. Namely, if the praxis directory contains a file named **options_*xxx*.sys**, where ***xxx*** is the program's label, it will be given precedence over ***options.sys***, i.e., it will be used instead.

Note that this feature requires no new parameters of **mkmk**. The multiple-program case is recognized by the script based on the presence in the praxis directory files named

[8] For legacy praxes only.

app_*xxx*.cc. If the directory contains such file(s) together with **app.cc**, the script will complain and generate no Makefile.

This mechanism works well if the multiple programs all fit the same board. This is because the single Makefile generated for the multiple program assumes the same board definition for all of them. This way the programs may share the same set of object files, i.e., the **KTMP** directory (or **temp** for CyanIDE). In many cases, however, the board definitions must be different for different programs.

5.2 Multiple Makefiles for multiple programs

In the situation when the multiple programs of the same praxis (stored in the same directory) must be compiled for different boards, you can create for them multiple Makefiles by invoking **mkmk** for each program in turn. This is accomplished by adding the program's label (the ***xxx*** characters from **app_*xxx*.cc**) as an extra argument for the script, e.g.,

```
mkmk WARSAW_ILS tag
```

In the above example, the script will use **app_tag.cc** as the program's anchor. The Makefile created by the script will be called **Makefile_tag**.

To provide for a smooth coexistence of the multiple programs and multiple Makefiles, their **KTMP** directories are also renamed by adding the respective labels (e.g., **KTMP_tag**) in the above case. For CyanIDE, this affects the directories **out** and **temp**. This way, the sets of object files contributing to the different images are completely separated.

Normally, as we mentioned before, each invocation of **mkmk** erases the volatile files and directories created by a previous invocation. For example, when you execute:

```
mkmk WARSAW_ILS
```

to create a *single* Makefile (for the possibly multiple programs of the praxis), the script will start by removing any Makefiles already present in the directory, as well as the **KTMP** directory (if it exists) and any existing **KTMP_xxx** directories previously created for multiple Makefiles. On the other hand, if **mkmk** is invoked with a label argument to create a program-specific (labeled) Makefile, it will *not* erase any labeled Makefiles or **KTMP_*xxx*** directories with different labels. This way, labeled invocations of **mkmk** are cumulative: they allow you to add new Makefiles to the praxis directory.

For illustration, consider the **eco_demo** praxis with three programs: Tags (collectors), Pegs (aggregators/satnodes) and custodians (special Pegs). These programs are anchored at files **app_tag.cc**, **app_peg.cc**, and **app_cus.cc**. You can create a single makefile for all threes programs,e.g., by running:

```
mkmk ARTURO_PMTH
```

but then only the collector program will be built correctly by the resultant **Makefile**. This is because different boards are needed for the remaining two programs. But you can do this instead:

```
mkmk ARTURO_PMTH tag
mkmk ARTURO_AGGREGATOR peg
mkmk WARSAW cus
```

which will yield three Makefiles: **Makefile_tag**, **Makefile_peg**, and **Makefile_cus**. By executing, e.g., **make -f Makefile_tag**, you will create the collector program (in files **Image_tag** and **Image_tag.a43**).

Note that each program can use its own (labeled) variant of **options.sys** (**options_tag.sys**, **options_peg.sys**, and **options_cus.sys**). If there is no labeled version for a given program, but the standard **options.sys** file is still present, it will be used by default.

This concept of multiple (labeled) Makefiles is compatible with object libraries, e.g.,

```
mkmk -M ../PMTHLIB tag
```

is a legitimate invocation of the script (assuming **../PMTHLIB** contains a valid library) and it does the same thing as the corresponding board-based invocation.

6 Argument syntax

Here is the formal argument syntax accepted by the script.

If the first argument does not start with a `-`, it is treated as a board name, i.e., it identifies a subdirectory in **PICOS/PicOS/MSP430/BOARDS** or **PICOS/PicOS/eCOG/BOARDS**. Otherwise, it must be either `-M` or `-P` indicating a library-based build (Makefile or CyanIDE project). In that case, the next (second argument) must specify the library path.

If the first argument is a board name, and the second argument starts with a `-`, then it can only be one of `-m`, `-p`, or `-l`. The first case specifies a Makefile build and is the default (i.e., `-m` can be skipped). The second case indicates a CyanIDE project (the board must fall under **eCOG**). The last case (`-l`) is for building an object library.

Unless `-l` has been specified, i.e., in all the remaining cases, one last optional argument, if present and not starting with `-`, is treated as a program label for multiprogram praxes.

One of these flags: `-W`, `-C`, `-L`, `-U` can occur anywhere in the argument list. It forces the specific format of file paths used by the script with mspgcc, which is otherwise determined automatically based on the location of mspgcc and the layout of its directory.

- `-W` Windows-style paths
- `-C` Cygwin-style paths, i.e., Unix paths used internally with Windows paths underneath
- `-L` Linux/Unix paths. This is the same as `-U`.

Normally, you never have to use those flags, unless your mspgcc has been misconfigured somehow. They are not applicable to CyanIDE (which only exists under Windows).

Sample usage:

```
mkmk CHRONOS -m watch
```

equivalent to:

```
mkmk CHRONOS watch

mkmk CHRONOS -l
mkmk -M ../CHRONOS_LIB watch
mkmk -P ../GEORGE_LIB
```

7 Cleanup

The script performs (intelligent) cleanups of known to it volatile components of the praxis directory (see pages 8, 12), i.e., those components that have resulted from its previous invocations and are going to be useless for (or perhaps could interfere with) the present invocation. The **Apps** directory contains a script named **cleanup**, whose role is to remove all volatile elements from all subdirectories of **Apps**, to revert the package to its globally *clean* state (e.g., before a GIT commit). This script should be invoked from **Apps** or from any of its subdirectories (at any level). In the latter case, it will confine its activity to the respective fragment of the package's tree.

The script traverses all the subdirectories from the place it has been called, and, in each subdirectory, performs the following actions:

- If the directory contains no file named **.mop**, it does nothing in that directory, although it still traverses all its subdirectories down to the leaves.

- If the **.mop** file exists, but is empty, **cleanup** executes the standard action described by the script **cleanapp** in Scripts. Here are the contents of that script:

```
rm -rf KTMP KTMP_* VUEE_TMP Image* *.cyp *.cyw cstartup.* \
      qqpack.asm *.lif irq.* internal.map out temp \
      out_* temp_* Makefile Makefile_* .gdbinit gdb.ini \
      *.o side.* side /LIBRARY .stackdump junk*
```

- If the **.mop** file is not empty, and its contents start with **#!**, **cleanup** executes that file as a script. That script is allowed to directly execute scripts located in the **Scripts** directory of the package.

- Otherwise, the **.mop** file is expected to contain a single line describing an invocation of a script from Scripts to be executed in the current directory.

Thus, any "standard" praxis directory should include an empty **.mop** file. If the cleaning requirements of the praxis are more subtle, it can put into **.mop** a detailed sequence of commands. For illustration, here are the contents of **.mop** from some project requiring a custom cleanup:

```
#!/bin/sh
cleanapp
./cleanup.sh
```

This means that, on top of the standard cleaning action prescribed by **cleanapp** (see above), a local script will be invoked (**cleanup.sh**) to handle the tricky bits.

Yet another global script related to clean-ups is **permissions** (in **Scripts**). When the package directories are moved back and forth across various (partially compatible) filesystems (say NTFS and Linux EXT3/EXT4), the permission bits of files and directories become messed up. To bring them back to order, you can execute **Scripts/permissions** in the root directory of the package. The script will traverse the

package tree setting file permissions based on their extensions and (reasonably well) guessed content.