Pawel Gburzynski

mkmk

the PicOS Makefile Maker

September 2021

Related referenced documents:

[picos]	Programming under PicOS
[pip]	PIP: an integrated SDK for PicOS
[vneti]	VNETI: Versatile NETwork Interface
[vuee]	VUEE: the Virtual Underlay Execution Engine
[picomp]	PiComp: the PicOS Compiler
[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

1 Preamble

This document describes the **mkmk** tool whose role is to create project Makefiles for compiling PicOS praxes into loadable images. The program is mostly hidden from the developer's view by PIP (the GUI SDK described in [pip]), so you may never have to delve into its internals. Nonetheless, this document will help you understand what exactly happens when a PicOS praxis is compiled. Note that **mkmk** takes care of "real-life" compilation, i.e., it produces images that can be loaded into real nodes. VUEE compilation [vuee] is handled by **PiComp** [picomp] (which, in one of its two basic modes, also happens to be invoked from Makefiles generated by **mkmk** as a preprocessor for the C compiler).

Note that **mkmk** was completely rewritten in February 2017, and modified a few times later on, as part of my effort aimed at accommodating other microcontrollers, specifically ARM Cortex (even more specifically, CC1350 by TI).[2] Support for the (now extinct) eCOG (Cyan) microcontroller and its platform was removed (that device is definitely dead now) which allowed me to simplify and streamline a few things. One reason for the rewrite was to make sure that whatever could be unified (regarding the two architectures available at present and, possibly, more to come) is in fact now unified and any differences are described clearly in some known, preferably single, and easily accessible place. So a few things related to porting should now be easier than they used to be.

2 The general idea

The program is typically executed in a directory containing the source code of a PicOS praxis [picos]. 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.[3] 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** will produce a Makefile constituting the input

[1] By Steve Underwood.
[2] See [armgcc] for the documentation of the ARM version of gcc and the associated toolchain.
[3] This is reminiscent of the standard UNIX tool **mkdepend**.

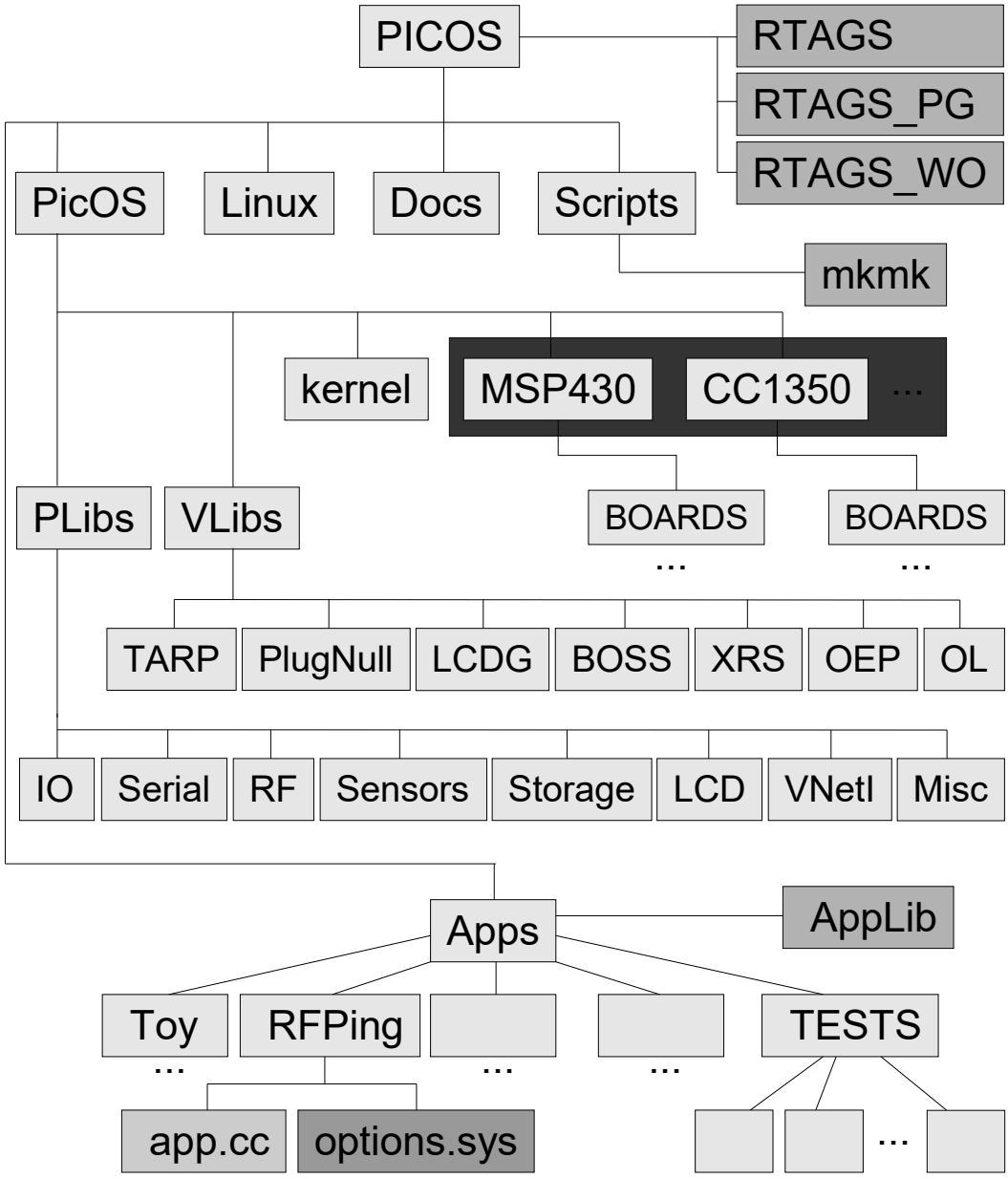

Figure 1: Source tree layout.

for **make**. Normally, the actual make operation (as well as the prior requisite invocation of **mkmk**) is carried out from **pip** in a GUI environment.

3 The principle of operation

The PicOS package comprises a hierarchy of files: kernel source files, device drivers, architecture-specific components, libraries, documentation, and so on. Application programs (called praxes) are also part of that hierarchy. Its skeletal and partial layout is shown in Figure 1.

A praxis to be handled by **mkmk** can be located anywhere within the filesystem hierarchy. Typically, for a project handled by PIP, the directory containing the praxis contains a PIP project. The praxes that come with PicOS as standard components of the package are stored in subdirectories of **Apps.**

The simplest and most popular usage of **mkmk** is to create a Makefile for a praxis. For that the program *must* be called in that praxis directory. The directory is expected to contain at least one source file of the praxis named **app.cc**.[4]

The general difference between files ending with **`.c`** and **`.cc`** is that the latter are assumed to contain PicOS code, i.e., additional PicOS-specific constructs requiring preprocessing into C by PiComp [picomp]. This is not C++. The former are plain C files.

Basically, there exist two ways to compile a praxis: the *source mode* (outlined below) and the *library mode* (discussed in Section 4), which requires the prior procurement of a requisite library.[5] The script will find out which other files must be compiled and/or 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 the source mode, the generated Makefile will prescribe the full compilation of all the contributing source files, i.e., the scheme assumes no precompiled, general purpose, object libraries, possibly except for the standard (non PicOS) libraries that come with the C compiler (or its toolchain). In our operation, this mode has been more popular, probably for historical reasons.[6] Our philosophy of parameterizing the system has favored a recompilation of its modules for the different sets of options. This is generally OK, because the system is small. Besides, once you run the Makefile generated by the program 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 **make** is run again.

3.1 Architecture, board, CPU type

Another source of input to **mkmk** (in addition to **app.cc**) is the target device (board) description which is contained in a special (board) directory. A praxis is always compiled for a specific board (more generally, for a set of boards, see Section 5). A board is described by a collection of standard header files which are **`#included`** by some system files and also, implicitly, by the praxis program (by being included from the standard header **sysio.h** needed by every praxis). Intentionally, board definition files refer to a specific model of the CPU with a specific set of devices, port mappings, 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 microcontroller *architecture*. At present, the architecture can be either **MSP430** or **CC1350**, but the structure is flexible and open for additions.

[4] Or, possibly, **app.c**. 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.

[5] Which action is also taken care of by **mkmk** (in a special extra mode).

[6] The primary culprit responsible for this approach is (was) 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 have departed from the CyanIDE-inspired model (retiring the eCog and CyanIDE support), the model is being reconsidered.

Generally, the idea is that the set of available architectures (as recognized by **mkmk**) is represented by those subdirectories of **PicOS** whose names consist exclusively of upper-case letters possibly intermixed with digits. Thus, for example, **PLibs, Vlibs**, and **kernel** (occurring as subdirectories of **PicOS** alongside **MSP430** and **CC1350**) will not be interpreted as architecture names (see Figure 1). Architecture directories are independent and separate in the sense that only one of them is used for compiling any given project. It is possible to have projects that can be successfully compiled for more than one architecture, but only one architecture takes part in any given compilation.

The minimum required set of arguments to **mkmk** includes a single argument viewed as the name of the directory containing the board description. For example, with this command:

```
mkmk CHRONOS
```

the program will try to determine the architecture as the name of that subdirectory of **PicOS** that: 1) follows the naming rules for architecture directories (see above) and 2) contains a subdirectory named **BOARDS/CHRONOS**. If the same board name occurs in more than one architecture, the invocation will fail and the architecture will have to be specified explicitly in this way:

```
mkmk CHRONOS -a MSP430
```

Note that generally it makes little sense to use the same board names in different architectures.

The program, as such, 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 program. 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 the features needed by the praxis, not all standard board headers may be needed. The obvious idea when preparing a new board description is to clone a reasonably close existing board and proceed from there.

Regardless how the architecture has been selected (implicitly or with -a), the selection determines: 1) a subset of system files to be compiled (the contents of the architecture subdirectory of **PicOS**), 2) the toolchain to carry out the compilation, linking, and so on. The latter is described by the file named **compile.xml** in the architecture directory. That file can be edited to specify/change the names of the toolchain components, select compilation/linking options, etc. The program makes minimal assumptions about the components of the toolchain which should make it easy to accommodate new architectures (with their specific toolchains) in the future.

While the architecture attribute of a build translates into a choice of some important subset of system files, another attribute, the CPU variant, may further parameterize the architecture-specific components. That attribute is located in the file **params.sys** which the program expects to find in the board directory. More generally, the file consists of text 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`** indicates the CPU model and **`-l`** specifies the list of system files to be compiled into a library for the library mode of **mkmk** invocation (as described

in Section 4). The **-l** parameter is optional and only needed if you 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 defaults to 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 known 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 (and basically useless) CPU model, assumed when the **params.sys** file is absent or it doesn't provide the model, is **msp430x148**.

For CC1350, a single CPU model is available at present and it is named **cc1350**.

3.2 The toolchain configuration file

Every architecture directory contains a file named **compile.xml** describing the way to invoke the compiler tools when building the praxis. That XML file is expected to define a single root element, <compile>. Below are the (slightly) simplified contents of that file for MSP430 illustrating all the elements relevant for **mkmk**.

```
<compile>
    <cc dospaths="yes">
        <define>bool=″unsigned char″</define>
        <define>SIZE_OF_AWORD=2</define>
        <define>SIZE_OF_SINT=2</define>
        <path>msp430-elf-gcc</path>
        <args>-mmcu=%cpu% -Os -g -Wundef</args>
        <args>-fno-strict-aliasing</args>
        <args>-I %ccdir%/../include</args>
    </cc>
    <ld dospaths="yes">
        <path>msp430-elf-gcc</path>
        <args>-mmcu=%cpu%</args>
        <postargs>-L %ccdir%/../include</postargs>
        <after>
            msp430-elf-objcopy %image% -O ihex %image%.a43
            msp430-elf-size -Ax %image%
        </after>
    </ld>
    <ar dospaths="yes">
        <path>msp430-elf-ar</path>
    </ar>
    <ranlib dospaths="yes">
        <path>msp430-elf-ranlib</path>
    </ranlib>
    <loaders>
        … not interpreted by mkmk ...
    </loaders>
    <vuee>
        … not interpreted by mkmk ...
    </vuee>
</compile>
```

The description refers to four tools of the chain: the compiler, the linker, the archiver (library creator), and the library indexer (ranlib). Only the first two tools are mandatory. The library tools are only needed for creating libraries.

One attribute common to the first four tools is “dospaths” interpreted when **mkmk** runs under Cygwin. Its role is to indicate whether the tool requires the DOS (Windows) format of file paths specified as its arguments (as opposed to the UNIX path format used by Cygwin-native tools). The default value of that attribute is “no” (the attribute is ignored under Linux).

The <path> element of a tool description shows the way to invoke the tool. The above file assumes that the tools are visible via the standard PATH. Full (UNIX-style) paths can be specified when this is not the case, although it is recommended to make the tools available directly by adding the respective directories to PATH.

The <args> element declares any special arguments needed by the tool. The program inserts some standard arguments (beyond those described with <args>). For example, when invoking the C compiler, it will assume that the compiler recognizes the standard arguments: –c, –o, and so on. The arguments declared with <args> will be inserted *before* the standard arguments. This holds for all the tools. If any special arguments should follow the standard arguments (which, e.g., may be the case for the linker), they can be declared with <postargs>. Multiple copies of <args> and <postargs> can appear within the same tool element. They are concatenated (with blanks inserted around them) in the order of their occurrence.

The optional <after> element specifies the sequence of commands to be inserted into the Makefile after the invocation of the tool. The commands are separated by newlines.

Some special sequences appearing within arguments and/or <after> inserts are substituted by **mkmk** in the following way:

%archdir% is replaced by the architecture name, i.e., the name of the architecture directory, e.g., MSP430, CC1350

%cpu% is replaced by the CPU variant as extracted from **params.sys** (see Section 3.1)

%ccdir% is replaced by the path to the directory that contains the C compiler executable

%image% is replaced by the root file name of the flashable image constituting the target of the build

The <define> element is only applicable to the C compiler and specifies any extra symbols to be defined (translating into –D arguments for the compiler). In principle, such defines can also be specified in <args>, but it makes some sense to keep them separate. Two symbols that should be defined there are: **SIZE_OF_AWORD** indicating the size in bytes of the integer type that can also store a pointer, and **SIZE_OF_SINT** indicating the size of the architecture's standard (preferred) integer type.

Any elements occurring in **compile.xml** but not expected (not interpreted) by **mkmk** are ignored. Normally, the file contains two such elements: <loaders> specifying the default configuration of flash loaders and debuggers (interpreted by **pip** [pip]), and <vuee>, specifying some architecture-specific definitions overriding the defaults of the VUEE emulator [vuee].

It is possible to provide **mkmk** with a non-standard toolchain configuration file by specifying that file explicitly in a call argument, e.g.,

```
mkmk CC1350_LAUNCHXL -a CC13XX -c toolconf.xml
```


The –c argument can appear anywhere on the list of arguments to the program and it must be followed by a file name. If the path is not absolute, it will be looked up first in the current (project) directory and then in the architecture directory (where the standard **compile.xml** file is located). The same applies to the standard file, so one way of overriding the toolchain configuration data is to copy the standard file to the project directory and edit it there. Normally, a non-standard toolchain configuration file is specified in **pip** where it can be consistently applied to all the components of the platform, including VUEE.

3.3 Standard #defines

The C compiler that will be invoked by the Makefile created by the program will receive some standard symbol definitions, quietly inserted by **mkmk** on top of the definitions brought in by **compile.xml** (see Section 3.2), which 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
```

These symbols identify the system version. As of the time of writing, the PicOS version is 5.4, so **SYSVER_U** will be defined as 5 and **SYSVER_L** as 4. ***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., **PG170721A**.

```
#define __aaaaaa__
#define __cccccc__
```

where ***aaaaaa*** and ***cccccc*** stand for the architecture name (e.g., MSP430, CC1350) and the CPU variant (e.g., msp430f148, cc1350), respectively.

```
#define BOARD_xxxxxx
#define BOARD_TYPE      xxxxxx
```

The two constants allow the praxis to know the board for which it is being compiled. The string **xxxxxx** in both cases stands for the simple 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 of the praxis are “made” independently

The above list only covers those symbols that are introduced 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. 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 example, note that **BOARD_TYPE** is set to a piece of text which is (usually) not defined as a symbol (i.e., it holds no value as such). 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**. The standard header **sysio.h** automatically provides a stringified version of the board type as:

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

3.4 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. There used to be 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 current (most recent) version numbers of the system and the revision tag (to be assigned to **SYSVER_U**, **SYSVER_L**, **SYSVER_T**) are obtained from the **RTAGS** files. The procedure (involving scanning all those files) used to be carried out by **mkmk** (essentially at every invocation), but these days the three components are precomputed at installation (by the **deploy** script [installation]) and made available via the **picospath** command (which **mkmk** executes to get them). The formal procedure of locating this information in the **RTAGS** files is carried out (at installation) as follows.

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 procedure selects the last tag (as textually occurring in the file) 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, the procedure examines all 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
```

This is the way to identify in RTAGS the points where a new version was released. 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 values following the tag are returned as **SYSVER_U** and **SYSVER_L**, respectively.

3.5 Collecting the files for compilation

The program determines 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 in the current directory, i.e., the directory in which **mkmk** has been invoked. The program 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 new files to be added.

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/cpuvariant"
"PicOS/architecture/BOARDS/target_board/"
"PicOS/kernel"
"PicOS"
"PicOS/PLibs/*"
"PicOS/VLibs/*"
"Apps/AppLib/*"
"."
"./*"
```

In the above list, ***architecture*** stands for the architecture directory name (e.g., MSP430, CC1350), ***cpuvariant*** is the CPU variant name extracted from **params.sys** (Section 3.1) 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 searched. For those subdirectories, the search order is alphabetical with respect to their global (normalized) path names.

Subdirectories **PLibs** and **VLibs** of **PicOS** store those system components that are not considered parts of the system core. **PLibs** is for hardware-specific (but CPU-architecture-independent) components, e.g., device drivers (PHYs, see [vneti]), while **VLibs** contains the higher-level (hardware-independent, CPU-architecture-independent) elements that can be shared by VUEE models (e.g., VNETI plugins, including TARP). The source files kept under **PLibs** have names ending with **.c** (plain C), while the names of the files stored under **VLibs** end with the suffix **.cc** (those files need to be preprocessed by PiComp). The **AppLib** subdirectory of **Apps** serves a similar purpose as **VLibs**, but is intended for application-related shared components (e.g., ones that can be added by the user).

Some subdirectories of the praxis directory are excluded from the search based on their names. One directory name always excluded from the search is *VUEE_TMP* where PiComp puts intermediate files for VUE[2] 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. This “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 program 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 "PLibs/IO/beeper.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 the 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 be compiled, even though that file is not explicitly **`#included`**. This may be necessary, because (usually) only **`*.h`** files are explicitly included, while some **`*.c`** or **`*.cc`** files may be needed to provide the actual code behind the headers. This is accomplished with a special C comment in this format:

```
//+++ "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`** and **`.c`**.

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, if the specification is ambiguous.[7]

3.6 Conditional processing

In addition to interpreting **`#include`** statements appearing in the parsed files, the program mimics (as accurately as possible) 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. The evaluation is somewhat heuristic but reasonably complete, although **mkmk** may not be able to authoritatively evaluate some legitimate expressions. For one thing, the program 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 program, which becomes known when the file is in fact compiled, may tacitly produce the wrong outcome.

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 program, you can insert this statement:

```
#define mkmk_eval v
```

in front of the respective condition. For as long as **`mkmk_eval`** remains defined, the program will ignore all conditions (not only the ones that cannot be otherwise evaluated) effectively assuming that they evaluate to ***v***, which should be 0 or 1 (if missing, ***v*** is assumed to be zero). You can revoke this setting by undefining the symbol, i.e.,

```
#undef mkmk_eval
```

Sometimes it makes sense to exempt an entire **`#include`**, e.g.,

```
#define  mkmk_eval
#include <some_header.h>
```

[7] 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 root name stored in **KTMP**. I may do something about this in the future, if it causes serious problems.

```
#undef   mkmk_eval
```

to prevent noisy and irrelevant complaints on some obscure headers.

3.7 The Makefile

At the end of an error-free run, the program will produce a Makefile (named **Makefile**) in the praxis directory, i.e., in the directory in which **mkmk** was executed. The program will also create a directory named **KTMP**. For every praxis source file whose extension is **.cc**, the Makefile will include an invocation of PiComp [picomp] to preprocess the file. The output from that preprocessing is stored in **KTMP** from where it will be compiled by the C compiler.[8]

Suppose we are looking at a source mode build. When you execute **make** in the praxis directory, the source files collected by the program 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 with a flash loader appropriate for the architecture/CPU variant.

Note that, for example, the <after> sequence in the linker description in **compile.xml** for MSP430 (Section 3.2) includes an invocation of **objcopy** to create an Intel HEX version of the flashable image (in addition to the standard ELF version). For CC1350, the ELF file is copied onto **Image.out** (the suffix is expected by some flash loaders recommended by Texas Instruments).

Whenever you invoke **mkmk** in a given praxis directory, the program starts by removing from it all files and directories that it thinks are the results of a previous **mkmk** or **make** run. In particular, it will remove the directory named **KTMP** and, in particular, the initializer file for the debugger, if one was created by a flash loader.

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, pin configurations, 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 VUEE selecting the optional library components (which role has been eliminated in PG121109A). Starting with R100217A, **options.sys** is optional. Note that for a multiprogram praxis the different programs may use different (specific) versions of the options file (section 5.1).

When using board libraries and compiling a praxis in the library mode (Section 4) a local **options.sys** file may become dangerously confusing and should be avoided in such cases, or perhaps strictly confined to VUEE by encapsulating its contents with:

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

[8] Legacy praxes (file names ending with .c) need no preprocessing of this kind: their source files are submitted directly to the C compiler.

(The constant is only defined when the praxis is compiled for VUEE [pip, vuee]). The confusion may arise because any constants defined in local **options.sys** and intended to modify some parameters of the kernel or a driver (e.g., the bit rate of the RF module) will be visible to the praxis files, but they will not affect the precompiled board library against which the object versions of those files will be linked. This may cause various errors ranging from linkage problems (if you are lucky) to a potentially malicious misbehavior of the praxis. In a nutshell, one should assume that local **options.sys** files (including multiple files for the different programs of a multiprogram praxis, Section 5) are incompatible with the library mode compilation. Note that the board-specific **board_options.sys** file should then authoritatively specify all the requisite parameters. A praxis willing to be compiled in the library mode against a different set of parameters should be compiled against a different library, i.e., a different board.

4 Library modes

A library mode makes it possible 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. Also, the procedure of compiling a praxis against a library is faster than compiling the praxis in the source mode.

4.1 Creating libraries

When called this way:

```
mkmk boardname [-a arch] -L
```

the program will create a Makefile for building an object library for the specified board. As for a standard source build, the architecture specification can be skipped, if the architecture is uniquely determined by the board. The Makefile created by the program is stored in the board directory.

An invocation of **mkmk** in the above format is not related to any project (no application/praxis files are needed) and can be issued from any directory. It will not affect the contents of that directory, unless it happens to be the directory of the target board.

The program reads **params.sys** in the board directory (Section 3.1) and expects to find there the parameter `-l` specifying the (space-separated) list of system files to be included in the library. These should be source files (names ending with `.c` or `.cc`). Any header files (`.h`) will be ignored. Note that all `//+++` requests (Section 3.5) issued from the source files (or from any header files included from those source files) will be honored (recursively) as for a regular source build.

At the and of a successful run, the program will generate a Makefile in the board directory. By running **make** in that directory you will create a file named **libpicos.a** containing the object library.

4.2 Compiling praxes against libraries

Once a library has been created, an alternative way to build a praxis for the board is to invoke the program this way (from the praxis directory):

```
mkmk boardname [-a arch] -l
```

The Makefile created this way will compile the praxis files only and link the resulting object files against **libpicos.a** in the board directory. No source files from the **PicOS** tree (or **Apps/AppLib**) are needed for that operation, but the relevant header files are.

When a praxis is compiled in the library mode, the inclusion of an object module from the library in the target executable is determined by the linker based (recursively) on external references from other modules. Some library modules, notably those providing interrupt service routines for MSP430, have to be indicated explicitly, because no module makes a formal external reference to the interrupt service function. This macro:

```
REQUEST_EXTERNAL (entry_point)
```

can be used to force such a reference. It expands into a linker directive (no code), so it can be put practically anywhere in the program. One standard application of this macro is to include the pin interrupt functions **p1irq** and **p2irq** for MSP430, e.g., see this fragment from the **board_pins.h** file for one of the boards for MSP430:

```
…
#define INPUT_PIN_LIST { \
      INPUT_PIN (P2, 2, 0), \
      INPUT_PIN (P2, 1, 0)  \
}
#define INPUT_PIN_P2_IRQ      0x06
//+++ "p2irq.c"
REQUEST_EXTERNAL (p2irq);
…
```

Although the special comment (see Section 3.5) makes sure that the file with the interrupt service routine will be included in source-mode compilation, the comment is obviously not seen by the linker when the module requesting the function comes in its object (compiled) version from a library. Thus, the role of **REQUEST_EXTERNAL** is to issue a formal external reference to the function, so that it can be found and included by the linker in the target executable. Note that this only concerns those functions that are never “normally” referenced as external in the code, i.e., interrupt service functions.

5 Multiprogram praxes and VUEE 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) flashable file that can be loaded into the target device (a board) of a single type. However, a praxis may consists of several different programs that may be loaded into boards of different types, e.g., data collectors (Tags) and access points (Pegs). 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 many advantages. The most important of them is the possibility to make all those programs available to VUEE as a single executable model in SMURPH/SIDE [vuee].

5.1 Single Makefile for multiple programs

As explained in 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, because it accommodates praxes consisting of multiple programs.

If the praxis directory contains no file named **app.cc**, but there is a 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 target image file built by the Makefile created this way will be named **Image_*xxx*** (the root ELF 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 will generate a Makefile to compile them all into separate images (**Image_one**, **Image_two**, and **Image_three**). A VUEE-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 entire praxis in VUEE [vuee].

When handling a program anchored at a labeled **app** file, **mkmk** changes a bit its procedure of locating **options.sys** (Section 3.8). 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., the labeled variant will be used instead and the unlabeled one will be ignored.

The multiple-program case is recognized by **mkmk** based on the presence in the praxis directory of files named **app_*xxx*.cc** (and requires no additional invocation options). If the directory contains such file(s) together with **app.cc**, the program will diagnose this as an error.

The mechanism, as described above, works fine if all the multiple programs fit the same target device (board), which is rather rare. 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. In most cases, however, the different programs are for different boards.

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, they can be built from multiple Makefiles created 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 program, i.e.,

```
mkmk boardname [-a arch] label
```

e.g.,

```
mkmk WARSAW_ILS tag
```

In the above example, the program will use **app_tag.cc** as the program's anchor. The Makefile created by the program 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 appending the respective labels (**KTMP_tag** in the above case).

In the single-program case (Section 3.7), 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 program 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: their effect is to add new Makefiles to the praxis directory.

For illustration, consider the old **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 three programs,e.g., by running:

```
mkmk ARTURO_PMTH
```

but then only the collector program will be built correctly from the resultant (single) **Makefile**. This is because different boards are needed for the remaining two programs. So you should do this instead:

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

which will produce three Makefiles: **Makefile_tag**, **Makefile_peg**, and **Makefile_cus**. By executing, say, **make –f Makefile_tag**, you will create the flashable image of the collector program (in file **Image_tag**).

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 as a fallback. If no applicable options file can be found at all, **mkmk** will assume that the program needs no options beyond those that come with the board (which is the recommended practice these days).

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

```
mkmk ARTHURO_PMTH -l tag
```

is a sensible invocation of the program (assuming that the board directory **ARTHURO_PMTH** contains a library).

6 Cleanup

The program performs (intelligent) cleanups of known to it volatile components of the praxis directory (Sections 3.7 and 5.2), i.e., those components that have resulted from its previous invocations and are going to be useless for (or perhaps could interfere with) the new invocation. The **Apps** directory contains a handy 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). When invoked in any directory, the program will clean up all praxes located in any descendants of that directory. This also applies to external praxes, i.e., ones stored outside **Apps**.[9] The program traverses all the subdirectories from the place it has been called, and, in each subdirectory, performs the following actions:

[9] But then the script must be referenced by its path in **Apps**. It may make sense to add to the praxis directory (or its superdirectory) a trivial script named **cleanup** containing that reference.

- 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 program **cleanapp** in **Scripts**. Here are the contents of that script:

```
rm -rf KTMP KTMP_* VUEE_TMP Image* Makefile Makefile_* \
      .gdbinit gdb.ini *.o side.* side .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 PicOS 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, **.mop** may contain 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 cleanups is **permissions** (in **Scripts**). When the package directories are moved back and forth across various (not 100% compatible) filesystems (say NTFS and Linux EXT4), the permission bits of files and directories tend to 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 heuristically guessed content.