Linux Software Build Guidelines

Version 0.7

Origination Date: 2008-02-13

Prepared By: Christopher A. Mosher

Confidential Material

TABLE of CONTENTS

1. REVISION HISTORY 4
2. DOCUMENT OVERVIEW 5
   2.1 Document Purpose 5
   2.2 Document Scope 5
   2.3 Document Location 5
3. REFERENCES 6
4. DEFINITIONS & ACRONYM LIST 7
   4.1 Definitions 7
   4.2 Acronyms 7
5. ADOPTING GNU MAKEFILE CONVENTIONS 8
   5.1 General Conventions for Makefiles 8
   5.2 Utilities in Makefiles 8
   5.3 Variables for Specifying Commands 8
   5.4 DESTDIR: support for staged installs 8
   5.5 Variables for Installation Directories 9
   5.6 Standard Targets 9
   5.7 Install Command Categories 9
6. MAKEFILE TARGETS 10
   6.1 all 10
   6.2 clean 10
   6.3 install 10
   6.4 check 10
   6.5 doc 10
   6.6 package 10
   6.7 iso 10
7. FILE NAME CONVENTIONS 11
8. PRODUCT BUILD PROCESS 12
9. BUILDING DEPENDENT COMPONENTS 13
   9.1 Overview 13
   9.2 topmake Script 13
   9.3 Running make 14
   9.4 Making the ISO image 15
10. COMPONENT BUILD PROCESS 16
   10.1 Process Overview 16
   10.2 File Requirements 16
      10.2.1 component.depends 16
      10.2.2 Makefile 17
      10.2.3 component.spec 18
11. SPECIFIC TYPES OF COMPONENTS 19
   11.1 FLEX Component 19
   11.2 PHP Components 19
   11.3 Product Component 20
   11.4 Scripts Component 20

11.5 C++ Application Component 21
11.6 C++ Static Library Component 21
11.7 C++ Shared Library Component 22
11.8 Java Component 22

12. THIRD PARTY SOFTWARE 24
12.1 Definition 24
12.2 Types of Third Party Software 24
12.2.1 Software with Suitable Binary RPMs 24
12.2.2 Software with Suitable Binary Distribution (no RPM) 24
12.2.3 Source-Only or Non-Suitable Software 24
12.2.4 Build Tools 25

13. WORKFLOWS 26
13.1 Creating a new build area 26
13.2 Doing subsequent builds 26
13.3 Building the RPM package file 26
13.4 Cleaning the build area 26
13.5 Building with topmake 27
13.6 Creating a new component and building it (tutorial) 27

1. Revision History

Rev	Paragraphs Changed	Purpose of Changes	Changed By	Date
0.1		Initial Release	Chris Mosher	2008-12-29
0.2		remove "dist" target; use $(TARGET) in topmake	Chris Mosher	2009-02-13
0.3		change "rpm" target to "package"; topmake does not default TARGET to rpm (but to blank instead); other minor corrections and clarifications	Chris Mosher	2009-02-23
0.4		change topmake TARGET to COMPONENT_TARGET (TARGET was already used in other Makefiles). Add Product Build Process/Building Dependent Components distinction; add ISO target; other minor fixes.	Chris Mosher	2009-03-02
0.5		change to buildtools handling; other minor fixes	Chris Mosher	

2. Document Overview

This document describes the software build process at IPC. It covers the actual building of software (make), and packaging (RPM), and related actions.

2.1 Document Purpose

The purpose of this document is to describe the software build environment used at IPC.

2.2 Document Scope

This document covers building software into deliverables. The build process applies only to the Linux platform. It does not cover configuration management (ClearCase).

2.3 Document Location

This document is stored in the SharePoint repository:

EngineeringStandards/Development/Make/SoftwareBuildGuidelines.doc

3. References

1. **GNU Coding Standards:** http://www.gnu.org/prep/standards/
2. **GNU Automake Manual:** http://www.gnu.org/software/automake/manual/
3. **Program Library HOWTO:** http://www.dwheeler.com/program-library/
4. **ClearCase Directory Structure:** http://newhome/TeamSites/TSProjects/EngineeringStandards/Development/SCM/New_CC_Dir_Structure.doc
5. **RPM Guidelines:** http://newhome/TeamSites/TSProjects/EngineeringStandards/Development/SCM/RPM_Guidelines.docx
6. **Linux FileSystem Hierarchy Standard (FHS):** http://www.pathname.com/fhs/

4. Definitions & Acronym List

4.1 Definitions

Component..... A component is one buildable entity. A component will be a library, an application, or a set of scripts, for example, that provides one set of functionality. A component will be delivered as one RPM package. A component will be buildable by the Makefile at its root level. The root level directory of a component is determined by the location of the *component*.depends file.

Product ..... A product is a system, for example, OneMS, ESS, EASe. It will consist of (and only of) a set of components. One of the components will be considered the top-level component, named after the product. The product-component will have as its dependents all the application components (at the highest level) that are within the product. That is what defines the product.

4.2 Acronyms

Acronym	Definition
FHS	Linux File System Hierarchy Standard http://www.pathname.com/fhs
GNU	GNU's Not Unix: http://www.gnu.org
RPM	RPM Package Manager: http://www.rpm.org
RHEL	Red Hat Enterprise Linux: http://www.redhat.com/rhel
VOB	ClearCase Versioned Object Base; a repository

5. Adopting GNU Makefile Conventions

In general, we follow the Makefile Conventions outlined in the GNU Coding Standards (http://www.gnu.org/prep/standards/). The Standard has a section entitled "Makefile Conventions" here: (http://www.gnu.org/prep/standards/html_node/Makefile-Conventions.html).

We will use GNU make for every component's Makefile. If possible, all building should be done in the Makefile. Specifically, we should avoid using omake, qmake, nmake, or *ad hoc* shell scripts, for example, to do any building of software. If, however, there are other tools that are more appropriate than GNU make for building (a limited area of) source code, then the Makefile can be written to pass-through to those tools. All effort should be made to use GNU make exclusively. The most notable exception is using Apache Ant for building Java code.

The remainder of this section mirrors the structure of the GNU Makefile Conventions document, and describes how we can adapt it for IPC's use. We depart from the GNU Coding Standards in some limited ways. The Standard warns about writing Makefiles that are compatible with all make programs; however, we will be using GNU make exclusively, so we can take advantage of any GNU-make-specific features in our Makefiles. Other differences are outlined in each section below.

5.1 General Conventions for Makefiles

We do not need "SHELL = /bin/sh" in the Makefile, because we are always using GNU make.

We will support VPATH builds. VPATH builds are described in the GNU Automake Manual here (http://www.gnu.org/software/automake/manual/html_node/VPATH-Builds.html). We only need to support VPATH builds, not normal builds.

Our Makefiles will not modify the source directory tree in any way (unless, of course, the build directory lies within the source directory, which is not normally the case). This is because only true sources will be checked into source control and will be a part of the distribution; we do not need to check-in or distribute any intermediate files.

We will support parallel make (the -j option).

5.2 Utilities in Makefiles

The shell used for the build will be bash.

The Makefile can use any tools that exist in the "developer" installation of the agreed-upon base platform (Note that for OneMS, the platform is RHEL 5.3), or approved 3rd party tools (see "Third Party Software" chapter, below).

We can use "mkdir –p" in our Makefiles; the build environment must support that.

Use make variables to locate compilers and linkers.

5.3 Variables for Specifying Commands

(No changes from the Standard.)

5.4 DESTDIR: support for staged installs

We must support DESTDIR as documented in the Standard.

5.5 Variables for Installation Directories

Each component will need to set the "prefix" variable. It will be set to /opt/ipc/*product*, where *product* is the name of the product, for IPC products, or /usr/local for third party components.

We do not use TexInfo for documentation.

5.6 Standard Targets

We will support the following targets as described in the Standard:

1. all
2. install
3. clean
4. check

We will support the following additional targets:

5. doc
6. package
7. iso

5.7 Install Command Categories

We will handle pre- and post-installation actions in the RPM package, not in the Makefile.

6. Makefile Targets

This section describes the various targets that our Makefiles will support.

6.1 all

Compiles all the component's source code to produce its program, library, etc. This is the default target of the Makefile. This will compile and link (if appropriate) all the sources for this component. Observe the GNU rules for VPATH builds when writing commands. Sources will be found through the VPATH variable; objects and other targets will get built into a "build" directory that gets created in the current default directory at the time the make command is run. Note that special handling is required for scripts or any other files that do not get compiled; they will need to be copied into the build area from the source area (if they don't yet exist, or are outdated).

All unit tests will be built, as well.

6.2 clean

Deletes everything that was built by the "all" target. If everything is built into the "build" subdirectory, then this target can simply remove that directory (recursively). Note: this does not remove products of the package target.

6.3 install

Installs the component into its final runtime location. The "all" target is a prerequisite for this target. Observe the GNU standards for the DESTDIR variable in order to support staged installs. This target would, for example, copy the component's executable program to $(DESTDIR)/$(bindir).

6.4 check

Runs the entire suite of unit tests for the component. The "all" target is a prerequisite for this target.

6.5 doc

Builds any and all documentation for the component. This Guildline does not go into any further detail on the doc target.

6.6 package

Builds a complete binary package for the component, in the platform's packaging format, which for RHEL is RPM. The package will be the only deliverable for the component to the run-time system. The RPM package will be named "COMP-VERS-1.ARCH.rpm" where ARCH is the target architecture (for example, i386). (See the dist target for a description of COMP and VERS.) This target has the COMP.spec file as a prerequisite. Note that creating the RPM winds up indirectly invoking make and calling this Makefile to build the "install" target (which in turn builds the "all" target).

6.7 iso

This target is defined for a product-level component only. It will assemble all the rpm files from the current set of components and create an ISO image file containing them.

7. File Name Conventions

Any file name processed by any Makefile or other build process can be composed ONLY of characters from this set of 7-bit ASCII characters:

1. Uppercase letters: A-Z
2. Lowercase letters: a-z
3. Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
4. Underscore: _ (Unicode 005F: LOW LINE)
5. Hyphen: - (Unicode 002D: HYPHEN-MINUS)
6. Period: . (Unicode 002E: FULL STOP)

No other characters are allowed. Specifically, no spaces, no parentheses, no accented characters, and no 8-bit "high ASCII" characters are allowed.

Component names must be unique, because, ultimately, these will become a (flat) list of RPM packages, which cannot contain duplicate names.

8. Product Build Process

The Product Build Process will build every component's package (e.g., RPM) for a product (see "Building Dependent Components" section), and then build the product's ISO image. It is designed to do as much work as possible for the build, and as such is the top level of the build process.

The Product Build Process is intended primarily for the SCM department, although it can still be used by developers if desired.

To use the Product Build Process, run make with the /usr/vobs/ipc_build/include/Makefile. You must specify the name of the product to build, by defining the PRODUCT make variable, for example:

```
$ make -f /usr/vobs/ipc_build/include/Makefile PRODUCT=onems
```

There are various other variables that control the Product Build Process that have convenient default values, but which can be overridden on the command line if desired.

The Product Build Process will perform the following actions:

1. Create the build area directory, which defaults to:

   ```
   /usr/vobs/ipc_product/build/BBBBBB.user/
   ```

   where BBBBBB is the BUILD_NUMBER [seconds since epoch], and user is the current user's username.

2. Read the following ClearCase attributes from the product's Makefile (/usr/vobs/ipc_product/PRODUCT/Makefile, where PRODUCT is the product specified on the make command line):

   a. VERS: The version number for the build (default value: "1.0").

   b. SCM_BASELINE: The baseline ClearCase label describing the source code for this build (default value: "UNKNOWN_BASELINE").

   These variables are passed, ultimately, to the individual component builds. If either of these attributes cannot be read, then the corresponding make variable will not be set by this makefile, so the component builds will use the default value shown.

3. Run topmake (see "Building Dependent Components," below) to create the necessary Makefile for the build.

4. Run make on that Makefile to build into the build area created in step 1, specifying PRODUCT as the target to build. This will build each component's package target (see "Component Build Process," below).

5. Run make on the product's Makefile, specifying "iso" as the target to build (see "Making the ISO Image," below).

See the comments in the /usr/vobs/ipc_build/include/Makefile for more details on which variable can be overridden to change the various locations specified in the steps above.

9. Building Dependent Components

9.1 Overview

The *topmake* script is responsible for building components along with their dependents (which are other components). The component being built may have other components it depends on, and those components may in turn have other components they depend on, and so forth. The topmake script generates a Makefile that embodies the hierarchy of component dependencies, thus allowing the user to build the component, first building all necessary dependents in proper sequence.

9.2 topmake Script

The process to build a component along with its dependents consists of a script, called *topmake*, that generates a Makefile that can build any or all of the product's components. The topmake script takes source root directories as arguments. The script will treat each of these directories as a tree containing *.depends files somewhere within it (nested arbitrarily deep). The script will search in each directory (and subdirectories of them) for all files named *.depends, and assume that each *.depends file it finds resides at the top level of a component's source tree. Furthermore, it will read the *.depends files to determine which other components each component depends on. Thus, topmake will determine the complete hierarchy of components for the product. For more information on the component.depends file, see the section "component.depends," below.

One of the components may be a "top level product" component, whose .depends file contains the top-level application components for the entire product. See the section "Product Component" below for more information on the product component. Building this product component would build every component in the product (but would not build the iso target of the product).

The generated Makefile will have one target for each component found. The rules for each component target will create a build area for that component and then recursively call "make" to build that component. The build areas for each component will be created in the current default directory.

Here is an example of running topmake to generate a Makefile:

```
$ cd /home/build
$ /usr/vobs/ipc_build/include/topmake /usr/vobs
.PHONY: onems

onems: bhc
   mkdir -p onems
   $(MAKE) -C onems -f /usr/vobs/ipc_product/onems/Makefile $(COMPONENT_TARGET)

.PHONY: bhc

bhc: cmdqued
   mkdir -p bhc
   $(MAKE) -C bhc -f /usr/vobs/ipc_admin/xyz/bhc/Makefile $(COMPONENT_TARGET)

.PHONY: cmdqued

cmdqued:
   mkdir -p cmdqued
   $(MAKE) -C cmdqued -f /usr/vobs/ipc_lib/cmdqued/Makefile $(COMPONENT_TARGET)
```

In this example, the user sets the current default directory to /home/build, which will become the *build area* for this particular build. All files resulting from this build will be within that directory. The argument to topmake in this case is /usr/vobs, which is the entire ClearCase repository (currently mounted using the current view). So topmake will search all of /usr/vobs for any *.depends files. In this example, it finds onems.depends, bhc.depends, and cmdqued.depends. (In a real product, of

course there would be many more components.) The generated Makefile contains targets and prerequisites that define the complete dependency hierarchy of all the components in the product. In this case, the product component is onems and it has one application component, bhc, which depends on one library component, cmdqued. (This is purely a fictional example.) The steps to build each component consist of creating the build area directory, such as /home/build/onems and /home/build/bhc, and then calling make recursively for the component's Makefile. The target for the components' Makefile calls can be overridden by defining the make variable COMPONENT_TARGET. For example, you can specify COMPONENT_TARGET=package on the make command line to build the "package" target of each component. Note that the recursive $(MAKE) commands read the Makefiles from each source directory where topmake found a *.depends file; in this case they are in various subdirectories under /usr/vobs.

The directory structure created within the current default directory for the build output is as follows:

```
./                          current default directory (BUILD_TOP points here)
   component1/                 build area for component1
      build/                      intermediates and targets of build
      rpm/                        rpmbuild work area (contains final .rpm)
   component2/                 build area for component2
      build/
      rpm/
   ...                         build areas for other components being built
```

All building happens within these directories. Source files are read from the source areas, but the source directories are never written to. In general, the build process does not write anything outside of the build area. This is to ensure that multiple builds could be run simultaneously on one machine. All Makefiles must be written to allow for this.

The build area will contain all the intermediate files and target files created during the build. In general, the build process leaves all temporary files around to aid in debugging of the build, or so future builds within the same build area will not need to recompile everything (only things that have changed).

Note that individual component Makefiles use the BUILD_TOP variable to refer to dependencies within other component areas, which will be at a well-defined location, regardless of the organization of the component directories within the source area.

9.3 Running make

To use topmake to do an actual build, you have a couple of choices. You can just pipe its output to make, for example:

```
$ cd /home/build
$ /usr/vobs/ipc_build/include/topmake /usr/vobs | make -f - someapp
```

This example calls topmake, just as in the above example, which generates the Makefile, then it pipes that Makefile to make using the "-f -" option, which tells make to read the Makefile from standard input.

Alternatively, you can send topmake's output to a file, and just reuse that file. However, note that if you do not re-generate this file, then any modifications to component dependencies will not be reflected in the makefile. However, if you keep that caveat in mind, you can generate the Makefile and have make just read that, for example:

```
$ cd /home/build
$ /usr/vobs/ipc_build/include/topmake >Makefile
$ make someapp
```

With this alternative, then, any time we want to re-build, we can just issue these commands to build, re-using the existing Makefile (keeping in mind the caveat noted above):

```
$ cd /home/build
$ make someapp
```

Or, even simpler, just this one command, no matter what our current directory is:

```
$ make –C /home/build someapp
```

The argument we pass to the make command is the target we wish to build, which is just the component name. In this case, we are building the someapp component. Additionally we could have set any make variables we wanted to on the make command line.

9.4 Making the ISO image

After running make to build all the RPM packages for the components, you can use the product-level component to make the ISO image. For example,

```
$ cd /home/build
$ mkdir onems
$ make –C onems –f /usr/vobs/ipc_product/onems/Makefile iso
```

This will set the default directory to the build area, and then run the product-level build for the onems product. By specifying the "iso" target, it will build the ISO image containing all the packages currently within the build area.

10. Component Build Process

Each component's source area has a root directory that contains a component.depends file, a Makefile, and a component.spec file. These files are described in the "File Requirements" sub-section of this section. The Makefile is responsible for building the component. It will not build any dependent components. The component's Makefile will be used by the topmake process (see "Building Dependent Components," above), by the Product Build Process (see above), and "stand-alone" by the developer manually running make.

10.1 Process Overview

The component build process will be a "VPATH build." A full description of VPATH builds is given in the GNU Automake Manual here (http://www.gnu.org/software/automake/manual/html_node/VPATH-Builds.html). Basically, a VPATH build implies that there are two primary directory trees:

1. the ***source*** *area*, and
2. the ***build*** *area*.

The *source area* contains all the component's source code, including the Makefile, component.depends, component.spec, and every source file needed to build to component. This area is not modified during the build process. The build just references it read-only. Typically, this would be within a ClearCase view, mounted under /usr/vobs, although that is not required; the source tree can, in fact, be anywhere. The root directory of the component's source area is what the VPATH make variable will point to; that's how make knows where to find the source files for the build.

The *build area* is where all the products of the build steps are placed. This would include object files (from C compiler), class files (from the Java compiler), swf files (from the Flex compiler), etc. Also, non-compiled targets, such as C include files, PHP files, or shell scripts, will get copied into the build area. The final product of the build, the RPM package file, will also be created in the build area.

The standard included makefiles define the VPATH variable as the *directory containing the component's Makefile*, so the actual defining of the VPATH variable is done automatically. The standard makefiles use the *current default directory* (at the time the make command is run) for the build area. Note that you can use the "-C" option of the make command to specify what the current directory shall be for than invocation of make; you do not necessarily need to us the "cd" command to set the current default directory.

10.2 File Requirements

This section lists files that are required in the root-level source directory of each component. There are example components of various types that demonstrate the build process. These are currently in ClearCase in ipc_build/examples. Templates are also provided; these are currently in ipc_build/templates.

10.2.1 *component*.depends

This is the most important file of the component, because it is this file that distinguishes this directory as a component that needs to be built by the topmake process and the Product Build Process. The *component*.depends file contains a simple list of the names of any components this component depends on. If this component has no dependent components, then the *component*.depends file will be empty, but still must exist nonetheless. Note that the developer is still additionally responsible for specifying, in the Makefile, any external dependencies that are needed for *compiling* or *linking* the component, and for specifying the dependent components in the Requires tag of the RPM spec file. Also note that the component's Makefile will not build any dependent components; instead, that is the responsibility of topmake (see "Building Dependent Components," above).

The file is named ***component*.depends**, where *component* is the name of the component. The format of this file is simply one component name per line, for example:

```
libutil
xdata_server
bhc
```

No other stray text or comments are allowed. This file will be expanded verbatim into the top level Makefile by the topmake script.

Note that the *component*.depends file is used only by the topmake script, and is not used by the component Makefile itself.

10.2.2 Makefile

The component's Makefile is responsible for:

1. Compiling and linking all of the component's source code into the target binary form, as necessary.

2. Installing the target binaries (scripts, etc.) into their final runtime directories (/opt/ipc/onems/bin for example), including support for staged installs via the DESTDIR variable as noted above in "DESTDIR: support for staged installs."

3. Generating a binary package (e.g., RPM file) for the component. This will be the only actual deliverable.

Standard Makefiles are defined that can simply be included by the component's Makefile. The standard Makefiles contain almost everything necessary, so the component's Makefile will generally be very small. The standard files are currently in ClearCase in ipc_build/include.

The following subsections describe the variables that must be defined in each component's Makefile, common to all types of components. For further information about the Makefiles for specific types of components, see the next section "Specific Type of Components."

10.2.2.1 Installation Prefix

You must define the prefix of the final installation directory path. This will (almost always) be one of the following:

```
/
/usr
/usr/local
/opt/ipc/product
```

(where product is the name of the IPC product).

For example, put this line in the Makefile for a component of the onems product:

```
prefix := /opt/ipc/onems
```

Do not include directories such as bin or lib in the installation prefix (it is just a prefix, not a full path). The build system will use the prefix to define standard variables, such as bindir and libdir, which will include both the prefix (such as /usr) and the directory (such as bin) (resulting in /usr/bin). These standard variables conform to the Linux File Hierarchy Standard (see references).

10.2.2.2 Toolchain Versions

Depending on the individual component being built, the toolchain may consist entirely of programs found in the base platform. However, if there are build tools that are not included in the base platform, then those tools must be installed onto the build machine first. See the "Third Party Software" chapter, below. The actual location and directory structure is defined in the /usr/vobs/ipc_build/include/tools.mk file. This file references each toolchain program, by specific version number variable. If your component needs to use one of these toolchain programs, then you must define the appropriate make variable to specify which version to use (no default value is provided). The tools.mk file documents which variables need to be used. For example, a flex component needs to specify which version of the flex compiler to use (and which version of java to run it with, because the flex compiler itself is written in java):

```
JAVA_VERSION := 1.5.0_16
FLEX_VERSION := 3.1.0.2710
```

10.2.2.3 Standard Include File

Each Makefile will then include the standard include file for the corresponding type of component (php, flex, c, etc.). This include is placed at the end of the Makefile, after any necessary variables are defined. You should define a variable for the root of the VOBs, in order to allow the location to be overridden on the command line. For example, a java component will have this in its Makefile:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/java_component.mk
```

10.2.3 *component*.spec

The rpmbuild process uses a file named ***component*.spec** (where *component* is the name of the component) to build the RPM. This will be a standard RPM spec file in some ways, but there are some restrictions imposed in order to work correctly with the standard build scripts. The main restrictions are not to use the Source tag because we are not starting from a .tar.gz source distribution; not to use "%prep" or "%build", only "%install" with "%ipc_make_install" macro; and to appropriately use the predefined symbols: component, VERS, BUILD_NUMBER, SCM_BASELINE, and the installation directories (bindir, libdir, ...). There is a standard template for the .spec file, currently in ipc_build/templates/component.spec file.

11. Specific Types of Components

11.1 FLEX Component

See the examples in ipc_build/examples/lib1flex and app1flex.

See template ipc_build/templates/flex.Makefile.

The directory layout of a FLEX-based component: there is a "src" directory with all FLEX source files, which are .as or .mxml files. There may also be a "test" directory with unit tests. In addition, in the top level must have a main class file, name *component*.as or *component*.mxml where component is the name of the *component*. There may also be other types of files in the src directory, such as assets like .jpg files or otherwise.

In the Makefile, you must specify the target of the build, which will be the .swc file for a FLEX library or the .swf file for a FLEX application. Usually you will want this to be the same name as the component, which can be achieved with the following definition:

```
TARGET = $(component).swf
```

(or .swc). Similarly, you must define MAIN_APP to be the main source file, such as:

```
MAIN_APP = $(component).as
```

The flex compilers will automatically compile and link in any extra classes that are referenced from the main class. These will be searched for under the src directory (or the test directory, for the unit tests). This may be all that is necessary for your application. However, there may be some classes that are not referenced from the main class that you still wish to compile and link in, for some reason (maybe for a library of classes that do not reference each other, for example). These classes can be added by listing them in the EXTRA_CLASSES make variable, for example:

```
EXTRA_CLASSES := com.ipc.onems.SomeUtils com.ipc.onems.AnotherThing
```

If the flex component has a dependency upon other components (which will be flex library components that generate .swc files), list them in the MXMLC_LIBRARIES variable (delimited with spaces), for example:

```
MXMLC_LIBRARIES := $(BUILD_TOP)/lib1flex/build/lib1flex.swc
```

Flex components must define JAVA_VERSION and FLEX_VERSION as the appropriate versions.

Flex components will include this standard include file:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/flex_component.mk
```

11.2 PHP Components

See the examples in ipc_build/examples/lib1php and app1ci.

See template ipc_build/templates/php.Makefile.

A PHP component will be either a library of PHP classes, or an application based on Code Igniter.

PHP components are a little different from other components, because PHP files are not compiled. However, they are still consistent with other types of components in that they need to generate output

files in the build area (for a VPATH build). In the case of PHP source files, we simply copy them to the build area. This is handled by the standard php include file:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/php_component.mk
```

If the component has unit tests to run with phpunit, then those tests will be placed in the "test" directory within the component, and must have a "AllTests" type of "suite" class that runs all the necessary unit test. In the Makefile, you must define the MAIN_TEST variable to the class name of this test suite class, for example:

```
MAIN_TEST := AllTests
```

And you must define the version of phpunit to use, for example:

```
PHPUNIT_VERSION := 3.3.9
```

There is a template directory to help in creating a CodeIgniter-based web application component. This template was taken from the CodeIgniter "application" directory, with some modifications. To create a new CodeIgniter based component, you will want to start by copying this entire template. The template is currently in ClearCase at ipc_build/templates/ci_app

11.3 Product Component

See the examples in ipc_build/examples/onems.

See template ipc_build/templates/product.Makefile.

Each product must have a single top-level component that defines (as its dependencies) the set of components that make up the product. The product.depends file will contain a list of top-level applications (or scripts or daemons, etc.) that make up the product. (Note that it only needs to list the top-level of dependencies, and does not have to list every library or other lower-level dependency, because the .depends files of the top-level applications will define those second-order dependencies, and so on for the entire hierarchy.)

The product component's .spec file will define a skeleton filesystem (empty directories) for the product, just so those directories get owned by a particular RPM). The .spec file must have a "Requires" entry that lists every component in the product (not just the top-level ones, but also every lower-level component). Of course, the spec file may define any other actions necessary for installation or removal of the product, as well.

The Makefile will just define the prefix variable, and include the standard product include file:

```
prefix = /opt/ipc/$(component)

VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/product_component.mk
```

11.4 Scripts Component

See the examples in ipc_build/examples/sys1scripts.

See template ipc_build/templates/scripts.Makefile.

A component may consist simply of a set of shell scripts. The scripts will all be in the component's top level directory, and will be installed into the bin directory under the prefix. This can be overridden by defining the INSTALL_DIR variable. There is a standard make include file:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/script_component.mk
```

11.5 C++ Application Component

See the examples in ipc_build/examples: cmdqued, app1cpp, app2cpp.

See template ipc_build/templates/cpp.Makefile.

A component that builds an application (or daemon) from C++ sources will need to define the TARGET variable as the name of the final application file, which should normally be the name of the component (but doesn't have to be), for example:

```
TARGET = $(component)
```

If the component needs to include and link against another component (a shared or static library), add the appropriate flags to CPPFLAGS and LDLIBS. When referring to the library component, use $(BUILD_TOP)/comp/build (where comp is the name of the other component). For example:

```
CPPFLAGS += -I $(BUILD_TOP)/xyz/build
LDFLAGS  += -L $(BUILD_TOP)/xyz/build –lxyz
```

The standard include file installs the application into the bin directory; this can be overridden in the Makefile by defining the INSTALL_DIR variable. For example, to install a daemon in the sbin directory instead of the bin directory, define INSTALL_DIR as follows:

```
INSTALL_DIR = $(DESTDIR)/$(sbindir)
```

Finally, include the standard include file for cpp application components:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/cpp_component.mk
```

11.6 C++ Static Library Component

See the example in ipc_build/examples/abc.

See template ipc_build/templates/liba.Makefile.

A component that builds a static library (.a file) from C++ sources will need to define the LIBNAME variable as the name portion of the library. It will usually be just the name of the component. For example, for a component named "abc" the static library will be named "libabc.a" so the name would be just "abc". For example:

```
LIBNAME = $(component)
```

The component will (almost certainly) have a header file (at least one) that defines the API for the library. Define the API_INCLUDES variable to list these header files (delimited with spaces). Often there will be only one header file named after the component, for example:

```
API_INCLUDES = $(component).h
```

Finally, include the standard include file for static library components:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/lib_a_component.mk
```

11.7 C++ Shared Library Component

See the example in ipc_build/examples/xyz.

See template ipc_build/templates/libso.Makefile.

A component that builds a shared library (.so file) from C++ sources will need to define the LIBNAME variable as the name portion of the library. It will usually be just the name of the component. For example, for a component named "abc" the shared library could be named "libabc.so.1.2.3" so the name would be just "abc". For example:

```
LIBNAME = $(component)
```

You also need to define the version number for the shared library. The version number consists of major and minor portions. For example, for version number "1.2.3a" the major version number is "1" and the minor version is "2.3a". The major version number must be just one number greater than or equal to 1 (and less than or equal to 20, as hardcoded in the rules_so.mk file). The minor version number may be anything (but cannot be empty). The major version number will be used in the SONAME for the shared library. For example, for shared library "libabc.so.1.2.3" the SONAME will be "libabc.so.1".

For example, for version number "1.2.3a" define LIBMAJOR and LIBMINOR as follows:

```
LIBMAJOR := 1
LIBMINOR := 2.3a
```

The major version number plays a special role for shared libraries. It defines a version of the API, such that increasing the major version number is an indication that the API has changed in a non-backward-compatible way. It normally requires users of the library to re-link against it to use the new version. This is achieved through the use of SONAMEs, and carefully named symbolic links in the filesystem. A full tutorial of shared library version numbers, their use, and their effects, is beyond the scope of this document. It is assumed that the developer of a shared library is fully aware of all aspects of shared library version numbers. A good introduction is the "Program Library HOWTO," by David Wheeler (see the references section).

The component will (almost certainly) have a header file (at least one) that defines the API for the library. Define the API_INCLUDES variable to list these header files (delimited with spaces). Often there will be only one header file named after the component, for example:

```
API_INCLUDES = $(component).h
```

Finally, include the standard include file for shared library components:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/lib_so_component.mk
```

11.8 Java Component

See the examples in ipc_build/examples/lib1java and lib2java.

See templates ipc_build/templates/java.Makefile and ipc_build/templates/build.xml.

For components written in Java, we use Ant to build the component. We still use make, but the Makefile will forward the "all" target processing to Ant, which will use the build.xml file to do the compiling.

In the Makefile, you will need to define the TARGET variable as the name of the final jar file, which should normally be the name of the component with ".jar" (but does not have to be), for example:

```
TARGET = $(component).jar
```

If the component depends on other components (jar files), then you define JAVAC_CLASSPATH to contain all these jar files (with paths in their final run-time location, but without a leading /). For example:

```
JAVAC_CLASSPATH := lib1java/build/lib1java.jar
```

Wildcards are allowed. Note that you do not need to include "$(BUILD_TOP)" in the names in the classpath; instead, the value of BUILD_TOP is passed into ant as the "build.top" property, and is used automatically when locating the classpath entries (as defined in the build.xml template file).

If there are JUnit unit tests, define the MAIN_TEST variable to be the name of the class that runs the suite of unit test. This class, and the tests themselves, are in the "test" directory of the component. For example:

```
MAIN_TEST := AllTests
```

Finally, include the standard include file for java components:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/java_component.mk
```

See the template build.xml file. The build.xml file will compile and jar the java source files. The template build.xml file may be able to be used unchanged in your java component.

12. Third Party Software

12.1 Definition

Third party software is any software that is not part of the platform (currently defined as RHEL, for OneMS) (which would be first party software) and not written by IPC (which would be second party software).

12.2 Third Party Software Installed with Product

Any software that is available from the official RHEL repositories is considered first party software, and is not a part of the build system (it will just be installed as RPMs). Software not in the RHEL repositories and not written by IPC can be categorized into different types, as follows.

12.2.1 Software with Suitable Binary RPMs

Third party software that is provided as a suitable binary RPM can simply be a part of the product installation, and does not have to be handled by the build system.

12.2.2 Software with Suitable Binary Distribution (no RPM)

Third party software that does not provide a suitable binary RPM, but provides a suitable binary distribution in some other form, such as a zip file, will need to be packaged up into an RPM by the build system. This will involve creating a new component (in ClearCase), and creating a .depends, .spec, and Makefile. The Makefile can include /usr/vobs/ipc_build/include/bin_component.mk, and will simply copy the provided binaries to the appropriate target directories (like /usr/local/bin for example). The prefix will always be /usr/local. When creating the component for third party software, the name of the component must include the full version number as specified by the software's authors. So as not to conflict with the version number used by our build process, we will add the component's version number as a part of the component's name, separated by an underscore. For example, if creating a component source directory for third party software Xyzap, version 3.45, name the component:

```
xyzap_3.45
```

The 3.45 is considered just part of the component's name, so when the build process creates the RPM package for this component, it will add the product build version to the name. For example, if building the above xyzap_3.45 component using our Product Build Process, specifying VERS=16.01.01b, the resulting RPM package will be named:

```
xyzap_3.45-16.01.01b-1.i386.rpm
```

which is the name of the component ("xyzap_3.45") followed by a hyphen, followed by the product version number ("16.01.01b"), followed by a hyphen, followed by the packaging number (always "1"), followed by a dot, followed by the architecture ("i386"), followed by the filetype (".rpm").

If we switch policy in the future to using specific version numbers for all our components instead of using the product version number, then this convention will become obsolete (and the version number to use for the third party component will simply be the version number of the third party software as specified by the authors).

12.2.3 Source-Only or Non-Suitable Software

Third party software that is only provided as source code, or whose binary distributions are not suitable for our use, will need to be built and packaged into an RPM by the build system. This will

involve creating a new component (in ClearCase), and creating a .depends, and either creating a .spec file and a Makefile, or modifying ones that are provided in order to meet our build requirements. Modifying third party files is not optimal, so if this is required it should be minimal and fully documented. For naming the component, follow the same naming convention described under "Software with Suitable Binary Distribution."

12.3 Third Party Software Used Only for Building

Third party software that does not need to be installed on the runtime system, but is only used at build-time, is handled differently from other third party software. These would normally be compilers, such as Sun's Java SDK, or Adobe's Flex SDK. These types of tools will need to be installed on any machine doing a build, via RPMs (which are either provided by the authors or written by IPC). The mechanism though with these RPMs are made available to developers is not specified in this document.

13. Workflows

13.1 Creating a new build area

If the source for a component exists, and the developer desires to build it, he/she can take the following steps to create a new build area and execute the build. For this example, assume the component is named "somecomp" and has a root source directory of /usr/vobs/ipc_lib/path/to/somecomp. Since you will most likely be running topmake (to build dependent components), you should set up a build area consistent with how topmake works. For example:

```
mkdir -p ~/build/somecomp
make -C ~/build/somecomp -f /usr/vobs/ipc_lib/path/to/somecomp/Makefile
```

This creates a fresh build area and sets your default directory there. Of course, you do not have to use ~/build/somecomp; the build area can be located anywhere. Since topmake creates subdirectories for each component, you will want to do the same; that's why we create a "somecomp" directory. Next, execute the make command. Use the -f option to specify which Makefile to use for the build. If the Makefile uses one of the standard build include files, VPATH will get set to the directory containing the Makefile. (Make will use the VPATH variable to locate any source files referenced in the Makefile.) The default target is "all," so this command will try to build the "all" target, which compiles everything and produces the target binaries. The default build Makefiles are set up to create a directory named "build" in the current default directory; the "build" directory will contain the intermediate and final binaries created during this step.

13.2 Doing subsequent builds

If the developer has already built the component into a build area, and later modifies some of the sources and wants to rebuild, he/she can run the make again from the same build area:

```
make -C ~/build/somecomp -f /usr/vobs/ipc_lib/path/to/somecomp/Makefile
```

This will rebuild any targets that are not already up to date.

13.3 Building the RPM package file

At some point, the developer will need to create a component.spec file, which specifies how to create the RPM package for the component. The default build Makefiles are set up so that the "package" target will create the RPM package file (for example, somecomp-1.0-1.i386.rpm) for the component. Building the RPM consists of simply specifying the "package" target to the make command:

```
make -C ~/build/somecomp -f /usr/vobs/ipc_lib/path/to/somecomp/Makefile package
```

The package target will create a directory named "rpm" in ~/build/somecomp and run the rpmbuild command to create the RPM file. The rpm directory is a workarea for the rpmbuild command. The process that rpmbuild goes through to create the RPM is complex. It involves (re-)running make with the install target, which first does the whole build (because the "install" target has the "all" target as a prerequisite), and then creates an image of the installed hierarchy (using the DESTDIR make variable). Finally, it creates the actual RPM file from that installed hierarchy. All this activity happens within the rpmbuild workarea (~/build/somecomp/rpm in this case). The resulting RPM package file will be: ~/build/somecomp/rpm/RPMS/i386/somecomp-1.0-1.i386.rpm.

13.4 Cleaning the build area

To "clean up" the build area, which consists of removing the generated target and intermediate binaries, the developer can simply delete the build area; for example:

```
cd ~/build/somecomp
rm –Rf *
```

Alternatively, the developer could (more safely) use the "clean" target of the Makefile, however the "clean" target only removes the "build" directory (the products of the "all" target), but will not remove any files created during the RPM creation process (the products of the "package" target).

13.5 Building with topmake

Once your component's source and rpm are building OK, or once you need to build dependent projects, you will want to use the topmake script to build your components. For example, assume your somecomp depends on othercomp component (whose source tree is somewhere within /usr/vobs/ipc_lib). You need to edit somecomp.depends and add a line with "othercomp" on it to the file. Then build both these components using topmake. For example:

```
cd ~/build
/usr/vobs/ipc_build/include/topmake /usr/vobs/ipc_lib | make –f – somecomp
```

This sets the current default directory to ~/build (this directory will contain a subdirectory for each component being built). Then it calls topmake passing in a directory containing all the source components we need to build (in this case, both somecomp and othercomp are somewhere within /usr/vobs/ipc_lib). Pipe the output of topmake to make, and specify "somecomp" as the target we want to build. Since our somecomp.depends file contains "othercomp", topmake will set othercomp as a prerequisite for somecomp. So, when we ask it to build "somecomp", it will first build othercomp, and then somecomp. The topmake script will build the default target of each component by default (the target can be overridden by defining the COMPONENT_TARGET make variable). See the "Product Build Process" section in this document for a description of variables you could pass to the make command. The VERS and SCM_BASELINE variables described there have acceptable default values (for a developer build), so we can omit them here. VERS defaults to "1.0", and SCM_BASELINE defaults to "UNKNOWN_BASELINE".

13.6 Creating a new component and building it (tutorial)

For this example, we will walk through the process of creating a new component. This example will make a very simple C++ application. The name of the component will be countargs. It will simply count the number of arguments passed to it, and print out the result.

First, we need an area to hold our sources. For this example, we will work outside of the ClearCase VOBs, and just use a normal local directory. You could, of course, work within the ClearCase VOBs, but for this example we want to focus on just the necessary actions for creating a component. (Of course, you will still need ClearCase access in order to copy the provided templates.)

```
$ mkdir –p ~/src/countargs
$ cd ~/src/countargs
```

Create the .depends file for our component. We have no dependent components, so our .depends file needs to be empty:

```
$ touch countargs.depends
```

Let's create the program. We name the file countargs.cpp, which is the same name as the component and the same name for the final executable we will be building. Keeping the names consistent makes it easier to track the component for others in the future. It also make it easier for make to use pattern rules. Create the file countargs.cpp as follow:

```
$ cat - >countargs.cpp
#include <iostream>
#include <ostream>
int main(int argc, char* argv[])
{
    std::cout << argc-1 << std::endl;
    return 0;
}
^D
```

Or use your favorite editor. Next we will create the Makefile so we can do a build. Copy the correct template Makefile for a cpp application from the template directory:

```
$ cat /usr/vobs/ipc_build/templates/cpp.Makefile >Makefile
```

(Using cat instead of cp ensures that the file is writable by us.)

Edit the template Makefile for our specific component. Replace the fields in the header area (marked with "???") for the name of the makefile, copyright, author, and creation date:

```
# Makefile for countargs
# Copyright (C) 2009, by IPC, Fairfield, CT
#
# author: Chris Mosher
# creation date: 2009-02-09
```

Let's assume we are creating a component for an IPC product named "foo"... we will need to change the prefix variable to reflect this:

```
prefix := /opt/ipc/foo
```

We can leave the definition of the target variable as is, because our target application is named "countargs", the same as our component.

```
TARGET = $(component)
```

We do not have any dependent libraries to add to the include path or the linker command, so we remove that whole section from the Makefile. And the default installation directory /bin (which translates to /opt/ipc/foo/bin) is fine, so we can remove those comments if we want. And the include mk file is correct for our component type:

```
VOBROOT := /usr/vobs
include $(VOBROOT)/ipc_build/include/cpp_component.mk
```

We now have enough to build our component. Let's create a new build area and try it out. Although not necessary, if we want to shorten our make commands, we can create a symbolic link in the build area that points back to the Makefile:

```
$ mkdir -p ~/build/countargs
$ ln -s ~/src/countargs/Makefile ~/build/countargs/Makefile
$ make -C ~/build/countargs
```

If all goes well, it should build without errors, and the results of the build should be in ~/build/countargs/build directory, including the final application. Let's run it:

```
$ ~/build/countargs/build/countargs a b c
3
```

The final aspect we will cover in this example is creating the RPM spec file. Get a copy the spec file template (which is provided in ClearCase):

```
$ cat /usr/vobs/ipc_build/templates/component.spec >countargs.spec
```

First , edit the "Summary" and "%description" areas provided with the template. We do not have any dependencies, so remove the "Requires" element, and the explanatory comment. The "%files" section is where we specify which files from our build get installed. The template provides some useful advice and many examples. For our case, with countargs, we only need to install the application file into the bin directory. With all these changes, the final countargs.spec file becomes:

```
Summary: counts arguments
Name: %{component}
Version: %{VERS}
Epoch: %{BUILD_NUMBER}
Release: 1
License: Proprietary
Group: Application
AutoReqProv: no

%description
%{SCM_BASELINE}
Prints number of arguments to standard output.

%install
%ipc_make_install

%files
%{_bindir}/*
```

We can test this out by running make with the "package" target:

```
$ make -C ~/build/countargs package
```

If all goes well, the RPM will be in rpm/RPMS/i386/countargs-1.0-1.i386.rpm. If you have enough privileges, you can install this rpm right onto your system, and run it:

```
$ sudo rpm -Uvh ~/build/countargs/rpm/RPMS/i386/countargs-1.0-1.i386.rpm
$ /opt/ipc/foo/bin/countargs foo bar
2
```