EPICS Source/Release Control

How to Build and Develop EPICS Software

Matthew Needes, Janet Anderson, and Marty Kraimer

PRELIMINARY DOCUMENT
February 16, 1995 (Draft)
EPICS Release 3.12

Table of Contents

Chapter 1: Introduction

1. Introduction to EPICS R3.12

EPICS source/release control has been completely redone for R3.12. This section describes why certain improvements were made and why they were worth waiting for.

Base and Extensions

EPICS was divided into two parts during the development of R3.12, “base” and “extensions.” Base includes those core parts of EPICS that are in general required, e.g.: channel access, database routines, the `sequencer` and `dct`. Extensions are those parts of EPICS that are optional. Unlike base, which is a fixed distribution, a site can choose which extensions they need for their local system. The extensions were unbundled from the base distribution to improve code maintenance and simplify the EPICS build.

The Build

The EPICS build has been simplified:

- `Imake`, and more importantly, `Imake.tmpl` are no longer used. Low-level makefiles are no longer automatically generated.
- A set of configuration files has been provided in a central location. Only a few of these files are site-modifiable, and are easily identified.
- The number of shell-scripts has been reduced from a set of over twenty to about ten. `Make` itself is now used to traverse the source tree during a build.

It is now required that GNU `make` version 3.71 or later be obtained for the EPICS build. Sun `make` is no longer sufficient.

Source Control, Development and Distribution

In R3.12, CVS has displaced SCCS as the tool for source control. CVS is superior to SCCS for several reasons, here are the most important ones:

- Operations can be performed on source trees instead of individual files.
- Entire revisions of EPICS, old or new, can be "checked out" with a given time stamp.
- Developers can concurrently edit the same files. CVS will interactively merge the changes later.

EPICS releases are maintained by a site designated as the "repository site." Though this site uses a tool for source control, EPICS is no longer distributed as a file repository. It is instead delivered as a set of files that do not have to be "checked out." This simplifies the build by removing a "`sccsInfo`" type step.

The most important result of using CVS for source control is that now EPICS software can be easily developed at separate sites concurrently. Sites that wish to develop parts of EPICS base must checkout a copy of EPICS `base` and `config`, and then create a release, noting both the time and date of this action. The release `Tar` file may then be taken to the local site and then merged into a local CVS tree. When a site is finished modifying EPICS, they can make a new release and merge it back at the master site. Specific instructions for this are provided later in this document.

Summary

During the development of R3.12 we have seen several changes in EPICS. CVS is now used instead of SCCS for internal source control, though the release itself is delivered as a set of straight files. Parts of EPICS are now unbundled products, such as the operator interfaces. But most importantly, the EPICS configuration has been greatly simplified, making EPICS easier to build at distribution sites.

Chapter 2: EPICS Introduction

1. The EPICS Release

An EPICS "release" is defined as a `Tar` file that contains the source code for EPICS `base` along with a set of configuration files (called "`config`") for building base and extensions. Since `base` and `config` are distributed in the same `Tar` file, the current EPICS revision number is associated with both of these directories.

The EPICS release number consists of a major, minor, and bug fix release number. In EPICS R3.12, for example, 3 is the major release number and 12 is the minor release number. Bug fix releases would be named R3.12.1, etc. A change in the major release number indicates a change in the channel access protocol, whereas a change in the minor release number indicates a minor change, such as an improvement to database processing. Since the channel access protocol does not change very often, it is usually possible to use older versions of unbundled extension software against newer versions of `base` and `config`. This is not always guaranteed, however.

Extension developers must provide release notes that indicate which versions of EPICS base their products are compatible with.

2. EPICS Base

EPICS base contains the following UNIX and VxWorks components:

- **atdb / dbta**: Short report to binary database translation (UNIX)
- **Channel access**: Communication layer between
- **dbLoadRecords/dbLoadTemplates**: Support for loading ASCII database files directly (VxWorks)
- **Record support**: The basic building blocks of EPICS applications (VxWorks)

- **Device support**: Glue layer between record and driver support (VxWorks)
- **Driver support**: Hardware interface (VxWorks)
- **State notation compiler**: Compiler for state-based programs (UNIX)
- **Sequencer**: Run-time environment for state-based programs (VxWorks)
- **ASCII files**: A set of files that describe EPICS databases and hardware
- **getrel / makesdr**: Application directory support (UNIX)
- **libCom**: Common libraries for base and extensions (UNIX / VxWorks)

3. EPICS Extensions

An EPICS extension may be obtained from the site that developed it. Here is a partial list of extensions and where to obtain them:

- **alh**: Alarm handler (ANL)
- **applSR**: Application source/release control system (ANL)
- **applSetupDir**: Simple application directory environment (LANL)
- **archiver**: Archiver and TCL/TK based data extraction tools (LANL)
- **burt**: Backup and restore tool (ANL)
- **ca** Simple channel access library and programs (ANL)
- **cau**: Channel access utility (ANL)
- **condition**: Programs for degaussing and standardizing magnets (ANL)
- **dalib**: Device access library (ANL)
- **dp**: Graphical Motif-based parameter display page (ANL)
- **edif2sr**: EDIF database representation to EPICS short report translator (LANL)
- **gvxi**: vxi support library (LANL)
- **km**: Knob manager (ANL)
- **knobconfig**: Knob configurer (ANL)
- **libUnix**: Unix utilities library (ANL)
- **math**: EPICS interface to Mathematica (ANL)
- **mdct**: Graphical Motif-based database editor (ANL)
- **medm**: Motif-based operator interface, OPI (ANL)
- **opi: edd / dm**: OPI editor and run-time environment (LANL)
- **orbitscreen**: Graphical Motif-based display of arrays of channel access data (ANL)
- **probe**: Graphical Motif-based display and modification of channel access data (ANL)
- **pvwave**: EPICS interface to PV-WAVE (ANL)
- **rampload**: Program for loading ramp tables into ramptable records (ANL)
- **sdds**: Self Describing Data Sets toolkit (ANL)
- **snaps**: Channel access data snapshot facility (ANL)
- **stripTool**: Channel access data stripchart tool (ANL)
- **wingz**: EPICS interface to Wingz (ANL)
- **xmca**: Motif-based X-window Monitoring of Channel Access data (ANL)
- **xmseq**: Motif-based run-time sequencer development environment (ANL)

4. Who to Blame

Jim Kowalkowski (ANL) and John Winans (ANL) developed the first set of `config` files to replace the original `imake` files. This first set provided support to build extension products for a single architecture. Janet Anderson (ANL) took over this responsibility and, working with the various tools developers, unbundled the tools that now reside in `extensions` (the original version used the term `add_on` products rather than extensions).

Matthew Needes (LANL) and Mike Bordua (LBL) then completely revised the `config` files in order to address portability concerns and support multiple architectures simultaneously in EPICS base. In addition they took time to greatly simplify the structure of individual makefiles as well as make them more consistent. The configuration files now support single host and multiple target architectures simultaneously. Janet Anderson made further changes to fix what Matt Needes broke in the extensions. She added support for make depends and added refinements to the make environment.

Matt Bickley and Johnny Tang of CEBAF developed the HP architecture specific `config` files. In addition, Pratik Gupta (CEBAF) made extensive changes in the source code, increasing its portability. Mark Anderson (ANL) developed DEC alpha/OSF configuration files.

Chapter 3: Directory Structure

1. The Master EPICS Tree

The EPICS tree consists of three top-level subdirectories: `base`, `config` and `extensions`. The `base` and `config` directories are distributed together in an EPICS release. Items within the `extensions` directory must be obtained separately.

- **epics/**
  - **base/**
  - **config/**
  - **extensions/**

NOTE: The files in the `epics/config` directory are used for building both base and extension products.

2. Base Sub-Tree

The `base` sub-tree contains the parts of EPICS that are generally required by all installations. It contains the top-level `makefile` used to build EPICS, subdirectories for executables, libraries, source, ASCII files and more.

- **epics/base/**
  - **Makefile**: EPICS top-level makefile.
  - **Makefile.subdirs**: Recurses through subdirectories. Used by `Makefile`.
  - **README**: Concise guide to building and developing EPICS code.
  - **bin/**
    - **sun4/**: `atdb`, `snc`, `bld*`, `antelope`, `e_flex`, more.
    - **hkv2f/**: `iocCore`, `vxWorks`, `recSup`, `seq`, more.
    - ...

- **include/**: EPICS header files.
- **lib/**
  - **sun4/**: `libca.a`, `libCom.a`, `libAs.a`, more
  - ...
- **man/**: Manual pages.
- **rec/**: Automatically generated header files, `default.dctsdr`, more
- **rec.bak/**: Old record header files from last run.
- **src/**: Contains source code and temporary objects.
- **tools/**: Shell scripts.

NOTES: Soft links to shell scripts are no longer maintained in each architecture's `bin` directory.

src Sub-Tree

The `base/src` directory contains a source sub-directory for each EPICS base Unix and vxWorks component.

- **epics/base/src/**
  - **as/**: Channel access security
  - **ascii/**: EPICS ASCII files.
  - **bld/**: Utilities for building SDR files in `epics/base/rec` and application directories.
  - **ca/**: Channel access client libraries.
  - **db/**: Database routines.
  - **dbtools/**: Libraries for loading ASCII database files.
  - **dct/**: Curses tool for generating EPICS databases.
  - **dev/**: Device support.
  - **devOpt/**: Optional EPICS device support.
  - **dr/v**: Driver support.
  - **iocCore/**: Directory in which `iocCore` is built from.
  - **libCom/**: UNIX/VxWorks common libraries.
  - **libCompat/**: Compatibility libraries for various architectures.
  - **libVxWorks/**: Various utilities for VxWorks IOCs.
  - **misc/**: Miscellaneous.
  - **rec/**: Record support.
  - **rsrv/**: Channel access server libraries.
  - **sequencer/**: State notation compiler and run-time support.
  - **toolsComm/**: Tools for building EPICS, e.g.: `e_flex` and `antelope`.
  - **util/**: Miscellaneous VxWorks and UNIX utilities.

Each source sub-directory contains source code, Makefiles and sub-directories for temporary objects. Here are the items you will see in a “typical” source directory:

- **epics/base/src/XXX/**
  - **Makefile**: Invokes the build in this source directory for all architectures defined in `CONFIG_SITE` by default.
  - **Makefile.Unix**: Makefile for UNIX software.
  - **Makefile.Vx**: Makefile for VxWorks software.
  - **O.<ARCH>/**: Automatically generated directories for temporary object storage. One directory for each architecture. (e.g: `O.sun4` or `O.mv167`) An `O.<ARCH>`

directory exists for each architecture being built in a particular source directory. `O.<ARCH>` and its initial contents are created when "`gmake dirs`" is invoked. Other dependencies such as "`gmake install`" will invoke "`gmake dirs`" for you if `O.<ARCH>` does not exist.

  - .**DEPENDS**: Automatically generated Include file dependencies. Empty if "`gmake depends`" has not been invoked.
  - **Makefile@**: Automatically generated symbolic link pointing to the build type Makefile (either `../Makefile.Unix` or `../Makefile.Vx`) corresponding to the current build type of this directories' target architecture.
  - **Target**.**Include**: Automatically generated file included by makefiles to indicate the architecture of this directory. This is necessary because Makefiles cannot easily determine the target architecture simply by looking at the current directory.
  - ***.o**: Temporary objects.
  - ***.a**: Libraries.
  - ***.c**: Automatically generated source (`yacc`/`lex`/etc.).
  - ***.h**: Automatically generated header files (`yacc`/`lex`/etc.).
  - *****: Executable files.
- ***.c**: C source files.
- ***.h**: C header files.
- ***.y**: Yacc files.
- ***.l**: Lex files.

Notes: The `O.*` directories and contents are generated by the build.

Tools Sub-Tree

EPICS shellscripts are found in the `base/tools` subdirectory of the EPICS distribution. Of these scripts, only gmake, getrel and makesdr should be invoked directly.

- **gmake**: Executes `gnumake` with the flags `no-print-directory` and `no-keep-going`.
- **getrel**: Sets up EPICS application directories by creating soft links to EPICS and vxWorks releases.
- **makesdr**: Creates a custom database template (`default.dctsdr`) for an application directory. This command is optional because a default template will be used if this command is not invoked.
- **MakeRelease**: Creates a Tar image of an EPICS release.
- **MakeDirs**: Creates temporary object directories under an EPICS release for a particular architecture.
- **GetVar**: Gets make variable information from within a shell script.
- **CheckArch**: Determines if configuration files exist for this architecture.
- **FindEpics**: Determines where EPICS is located from within shell script. Many shell scripts invoke this shellscript first.
- **Clean**: Removes temporary objects from the `O.ARCH` directories under `src`. If `Clean` is called without any arguments all `O.ARCH` directories are removed.
- **makeStatTbl**: Creates the error symbol table.
- **blderrSymTbl**: Builds the error symbol table.
- **SetOpVersion**: Selects specific executable versions of EPICS extension programs that will be used on future command invocations.

3. Config Sub-Tree

The configuration directory contains a set of files containing variable and rule definitions used by makefiles to build both base and the extensions. Files with names containing the word "SITE" are meant to be edited by the EPICS system manager only. The `CONFIG` file may be edited by both individual developers as well as the system manager. However, only the system manager may commit changes to `CONFIG`.

NOTE: Individual developers should never edit any file except CONFIG.

- **CONFIG** - This file is indirectly included by EPICS low-level Makefiles. It allows individual developers and the EPICS system manager to override specific variables in the configuration. This file conditionally includes other EPICS configuration files depending on the architecture being built.
- **CONFIG_SITE** - This file contains top-level variables that control parts of the build. Only the EPICS system manager is permitted to edit this file. It specifies the following variables:
  - **HOST_ARCH**: Your system's host architecture. This is the architecture that is used to build databases, run dm, etc. This may be set to `sun4` or `hp700`, for example. Only one architecture is allowed on this list at a time. A list of valid choices is provided in a comment above this option.
  - **BUILD_ARCHS**: List of all workstation and IOC architectures to be built for your local site. `HOST_ARCH` must be first on this list. A list of valid choices is provided.
  - **VX_DIR**: The location of your VxWorks distribution.
  - **EPICS_BASE**: Location of the official EPICS base release from which `extensions` will obtain base libraries, header files, etc.
  - **EPICS_EXTENSIONS**: Location of the official EPICS extensions release from which `extensions` will obtain base libraries, header files, etc.
  - **ANSI**: The C compiler you want for ANSI compilations. Choose the compiler from the list of uppercase options.
  - **STANDARD**: The C compiler to use for old-style C compilation. This option exists because not all parts of EPICS may be compiled under ANSI C (yet). Choose the compiler from the list of available uppercase options.
  - **CPLUSPLUS**: This variable is for future use. Ignore for now.
  - **STATIC_BUILD**: `YES`/`NO` - Whether or not to build UNIX programs statically.
  - **UNIX_OPT**: `YES`/`NO` - Whether or not to use the `-O` option when building UNIX objects.
  - **VX_OPT**: `YES`/`NO` - Whether or not to use the `-O` option when building VxWorks objects.
  - **UNIX_WARN**: `YES`/`NO` - `NO` suppresses warnings with the `-w` option, `YES` runs the compiler without additional warning flags.
  - **VX_WARN**: `YES`/`NO` - Same as above, but for VxWorks.

NOTES: The `VX_WARN` and `UNIX_WARN` options are ignored during a `STRICT` compile. `STRICT` compilation automatically uses the most verbose warning flags available. Compile types such as `STRICT` are set in the low-level makefiles. See the "Build Internals" section for more information about compile types such as `STRICT`.

- **CONFIG_SITE.Unix.ARCH** - This file specifies where to find directories for various libraries such as X, MOTIF, WingZ, etc.. If you do not have a certain product, leave the line empty. These directories are used by the extension products. `CONFIG_SITE.Unix.ARCH` contains the following variables:

- **CMPLR**: `STRICT`/`ANSI`/`OLD` - Default to `ANSI`. For 3.12, set this to `ANSI`. In the future this option may be more useful.
- **INSTALL_PRODUCT** - Path to program used to install extension products
- **SPECIAL_LANG**: Special directory to search for SUN patch libraries.

The following are path variables for various products. If you do not have a certain product, leave its line empty.

- **X11_LIB**: Path to X11 libraries.
- **X11_INC**: Path to X11 header files.
- **MOTIF**: etc.
- **OPENWIN**:
- **WINGZ**:
- **MATHEMATICA**:
- **XRTGRAPH**
- **QUESTWIN**:
- **IDL**:

The remaining files in the `config` directory are described in the "Build Internals" portion of this document.

4. Extensions Sub-Tree

The `extensions` sub-tree contains parts of EPICS that are not generally required by all installations. An individual site usually has only a subset of all available extensions. A complete `extensions` sub-tree contains a top-level `makefile` that can be used to build all extensions.

- **epics/extensions/**
  - **Makefile**: Extensions top-level makefile.
  - **bin/**: Installed executables and scripts
    - **sun4/**: `AR`, `alh`, `cau`, more.
    - **hkv2f/**: ...
    - ...
  - **include/**: Installed header files used by multiple extensions.
  - **lib/** Installed libraries used by multiple extensions
    - **sun4/**: `libUnix.a`, `libLca.a`, `libdev.a`, more
    - **...**
  - **man/**: Installed manual page files.
  - **doc/**: Installed documentation files.
  - **src/**: Contains source and temporary objects.
  - **...**

src Sub-Tree

The extensions source tree contains all the source code, header files, scripts, man page files, makefiles, and all other files for EPICS extensions. Each extension directory contains all the required files for that extension.:

- **epics/extensions/src/**
  - **alh/**: Alarm handler.
  - **ar/**: Achiver and TCL/TK based data extraction tools.
  - **burt/**: Backup and restore tool
  - **gdct/**: Graphical Motif-based database editor
  - **opi/**: OPI editor and run-time environment
  - **km/**: Knob manager
  - **medm/**: Motif-based OPI
  - **...**

Each extension subdirectory also contains Makefiles and directories for temporary objects. Here are the files you will see in a "typical" source directory:

- **epics/extensions/src/XXX**
  - **Makefile**: Invokes the build in this source directory for all architectures defined in `CONFIG_SITE` by default.
  - **Makefile.Unix**: Makefile for UNIX software.
  - **Makefile.Vx**: Makefile for VxWorks software.
  - **O**.*/: Directories for temporary object storage. One directory for each architecture. (e.g: `O.sun4` or `O.mv167`)
    - **.DEPENDS**: Automatically generated header file dependencies.
    - **Makefile@**: Symbolic link to either `../Makefile.Unix` or ../`Makefile.Vx`.
    - **Target.Include**: File included by `Makefile` that contains the target architecture definition.
    - ***.o**: Temporary objects.
    - ***.a**: Libraries.
    - ***.c**: Automatically generated source (`yacc`/`lex`/etc.).
    - ***.h**: Automatically generated header files (`yacc`/`lex`/etc.).
    - *****: Executable files.
  - ***.c**: C source files.
  - ***.h**: C header files.
  - ***.doc**: Documentation files.
  - **man***: Man page files.
  - ***.y**: Yacc files.
  - ***.l**: Lex files.

Notes: The `O.*` directories and contents are generated by the build.

Chapter 4: Getting Started

1. Introduction

In order for the source/release control commands to work, you must have your path and environment variables defined properly. The value of these variables is different depending on whether you are a user of EPICS or an EPICS developer and which part(s) of EPICS you are using. An EPICS developer is also considered to be an EPICS user and, therefore, must have all user paths/variables properly defined in addition to applicable developer paths/variables. Proper settings for path and environment variables will provide the following capabilities:

- **USER**
  - BASE
    All base components are automatically located relative to the directory in which getrel is executed. This allows the user to work on multiple EPICS base releases without changing environment variables.
  - EXTENSIONS
    The latest version of each extension product that has been installed by the local EPICS system manager is automatically executed. The user is also able to selectively choose other versions of any extension product.
  - MAKE
    Users will have gmake automatically located from their path. Gmake is a script that executes `gnumake` (the GNU version of `make`).
- **DEVELOPER**
  - BASE and EXTENSIONS
    All base components are automatically located relative to the directory specified in the developer's <epics>/cshrc file. This allows the developer to easily change environment variables and work on multiple EPICS base trees.
  - EXTENSIONS
    The developer may have a private extension tree containing his/her extension products only. By letting `EPICS_EXTENSIONS_PVT` reference the developers tree,

the developer can use his/her own products while still using all the other installed extension products.

The remainder of this section describes, for `csh` users, how to create an environment that provides the above capabilities. Similar methods can be used for other Unix shells.

2. User Environment/Path Requirements

.cshrc

The EPICS system manager provides a file which EPICS users can source from their .cshrc file to satisfy all the EPICS user Unix environment and path requirements. Ask your local EPICS system manager for the location and name of this file.

At APS, an EPICS user needs to place the following statement in his/her .cshrc file after any set path statements.

```
source /usr/local/etc/Cshrc.aps
```

Private extensions

If the user has a directory containing his/her private versions of EPICS extension products only, setting `EPICS_EXTENSIONS_PVT` to this directory prior to the above statement will allow the use of his/her own products while still using all the other installed extension products.

```
setenv EPICS_EXTENSIONS_PVT <private extensions>
source /usr/local/etc/Cshrc.aps
```

`<private extensions>` is the location where the user wants to pick up private versions of extension products.

3. Developer Environment/Path Requirements

.cshrc

Each EPICS base developer should add the following statements to his/her `.cshrc` file.

```
set oldpath = ($path)
setenv CVSROOT <cvsroot>
```

Ask your local EPICS system manager for the location of `<cvsroot>`.

epics/cshrc

This file is needed by both `base` and `extension` developers. This file must be created by the developer and placed in the developers private EPICS directory. The file must be executed whenever the user wants to use that EPICS development tree.

For a developer of `base` or a developer of both `base` and `extension`, this file contains the commands:

```
set DVL = `pwd`
set path = ($DVL/base/tools $DVL/base/bin/`arch` $oldpath)
rehash
```

For a developer of `extension` only, this file contains the commands:

```
set REL = <epics>
```

```
set path = ($REL/base/tools $REL/base/bin/`arch` $oldpath)
rehash
```

`<epics>` is the location of an official EPICS release with which the extensions will be built.

Chapter 5: User Interface

1. Introduction

This chapter explains how to create an application that uses base products. This normally means IOC applications. Channel access client extensions can also be executed in this environment.

The environment described here is fine for small applications and for testing. If you have a large application there is an extension product (Application Source/Release Control System) that provides a much more complete environment.

2. Creating an Application Directory

Execute the following commands:

```
mkdir <appdir>
cd <appdir>
```

You will need a vxWorks startup script file. The following is a sample:

```
# vxWorks startup script to load and execute system (iocCore) software.
ld < targetmv167/iocCore
ld < targetmv167/drvSup
ld < targetmv167/devSup
ld < targetmv167/recSup
ld < src/O.mv167/myobj1.o
ld < src/O.mv167/myobj2.o
dbLoad  "default.dctsdr"
dbLoadRecords  "<app1>.db"
dbLoadRecords  "<app2>.db"
iocLogDisable=1
iocInit
```

`<app1>.db` and `<app2>.db` are database files created in `gdct` database format.

3. Getting a Release

If this is a new application or if you want to use a new base release execute the command:

```
<epics_path>/base/tools/getrel <epics_path> <iocarch> [<iocarch>]
```

where <epics_path> is the location of an EPICS tree. For example if you want release 3.12 and you have an `mv167` IOC processor then issue the command:

```
/usr/local/epics/R3.12/base/tools/getrel /usr/local/epics/R3.12 mv167
```

After issuing this command issue the `ls` command. You will see that `getrel` created several soft links for you. These links allow you to:

1. Create, load, and execute IOC databases and sequence programs,
2. Create and use private record/device/driver support,
3. Create and use other IOC software.

You can also build Unix code in this environment but you might prefer to do this in an extension development environment. Included are the soft links:

- **ascii** - This is a directory containing all the ASCII files that are part of the release.
- **base** - This is the base directory tree for the release.
- **config** - This is a soft link to the config files used to generate the release.
- **default.dctsdr** - The file used by database configuration tools and by the IOC for obtaining the information contained in the ascii files. Note that if you run `makesdr` (see the next section) this soft link will be replaced by a hard file.
- **default.sdrsum** - A file containing checksum output from the command: "`sum default.dctsdr`".
- **include** - A soft link to the include files that belong to the release.
- **targetmv167** - This is a soft link to the base/bin directory for the `mv167`. Similar links are developed for each architecture requested by the getrel command.
- **vxWorksmv167** - A soft link to the vxWorks image for the `mv167`.

4. Boot Parameters

The following are sample IOC boot parameters.

```
boot device          : ei
processor number     : 0
host name            : <boot host name>
file name            : <full path name to your app dir>/vxWorksmv167
inet on ethernet (e) : <inet address of ioc>:ffffff00
inet on backplane (b):
host inet (h)        : <inet address of host>
gateway inet (g)     :
user (u)             : <userid for ioc>
ftp password (pw) (blank = use rsh): <password for ioc>
flags (f)            : 0x0
target name (tn)     :
startup script (s)   : st.cmdmv167
other (o)            :
```

5. Creating Private ASCII Files

The `getrel` environment allows you to create private record/device/driver support as well as override the support provided with a release. The first step is to create two directories:

- **replace_ascii**
  This directory contains any ASCII files, supplied with a release, that you want to completely replace. PLEASE THINK HARD BEFORE DOING THIS. Normally you should not replace EPICS supplied support. Note that `getrel` created a file "`ascii`" which is a soft link to the ASCII files supplied as part of a release.
- **cat_ascii**
  This directory contains two types of files: The first is files that should be concatenated to the end of the corresponding file located via the `ascii` soft link. The second type of file is new ASCII files such as files defining new record types.

As an example let's say that you want to create:

1. A new record type called `myRecord`.
2. Device support for the new record called `devMyRecord`.
3. A new driver called `myDriver`.

The following files are placed in `cat_ascii`:

- **myRecord.ascii**: This file contains the definition for the new record type.
- **dbRecType.ascii**: This file contains the line:
  `"my"`
- **devSup.ascii**: This file contains a line defining the device support. For example:
  `"my"    VME_IO      "devMy"     "My Private Device Support"`
- **drvSup.ascii**: This file contains the definition for the new driver. For example:
  `"drvMy"`

After you have created or modified the local ASCII files (or after a new `getrel`) then just execute the command:

```
makesdr
```

This command creates the file `default.dctsdr` which contains the information from all the ASCII files including yours. It also creates a directory `sdrH/rec`, which contains all the record type header files.

6. Building Private Source Files

If you want to develop private components for use on Unix and/or in an IOC (record support, etc.) then it is a good idea to create a subdirectory called `src`. It is then possible to create makefiles that use `base` development configuration files. This document describes these configuration files in Chapter 8, Section "Configuration Files" on page 37. The following samples are intended to show how simple makefiles can be. The following makefiles assume that they and all the source files reside in a directory one level below where `getrel` is executed.

Components for all site supported architectures will be built in automatically generated `O.<arch>` subdirectories via the command "`gmake`" from the `src` directory.

The Makefiles for building the private source files have the same structure and features as the Makefiles described in Chapter 6, Section "Using EPICS Makefiles" on page 24 and Chapter 6, Section "Creating High-level Makefiles" on page 25, except that the default rule has been set to "`build`" instead of "`install`" because "`install`" is not meaningful when in an application directory.

Makefile

```
EPICS=..
include $(EPICS)/config/CONFIG_BASE
default: build
include $(EPICS)/config/RULES_ARCHS
```

Makefile.Vx

Execution of this Makefile is from `O.<arch>` subdirectory.

```
EPICS=../..
include Target.include
include $(EPICS)/config/CONFIG_BASE

USR_INCLUDES = -I$(EPICS)/replace_ascii -I$(EPICS)/cat_ascii \
        -I$(EPICS)/sdrH/rec -I$(EPICS)/include/rec
SRCS.C = ../<src1>.c ../<src2>.c ../<src3>.c ../<src4>.c
TARGETS = <src1>.o <src2>.o <src3>.o <src4>.o

default: build
include $(EPICS)/config/RULES.Vx
```

Makefile.Unix

Execution of this Makefile is from the `O.<arch>` subdirectory.

```
EPICS=../..
include Target.include
include $(EPICS)/config/CONFIG_BASE

DEPLIBS = $(EPICS_BASE_LIB)/libca.a $(EPICS_BASE_LIB)/libCom.a
USR_LDLIBS = -lca -lCom
SRCS.c = ../src1.c ../src2.c ../src3.c
OBJS = obj1.c obj2.c obj3.c
TARGETS = target

default: build
include $(EPICS)/config/RULES.Unix

target:  $(OBJS) $(DEPLIBS)
    $(RM) $@
    $(LINK.c) -o $@ ${OBJS} $(LDLIBS)
```

Chapter 6: Developer Interface

1. Introduction

This chapter explains how to create and use a private master EPICS directory tree for developing EPICS base and extension software. The CVS source/release system makes it convenient to develop for base and for individual extension products or for a combination.

2. Creating Private Directory Structure

Checkout 'config'

```
cvs checkout epics/config
```

This will checkout the configuration files used by both base and extensions. `config` must be checked out for any development work on base and/or extensions. It creates a subtree named `epics/config` in whatever directory the `cvs` checkout command is issued.

Please note that the only configuration file a developer should ever modify is the file `<epics>/config/CONFIG`. You can add definitions to the end of this file to override definitions contained in other config files.

Checkout 'base'

Checkout `base` only if you wish to develop `base` products. Although it is possible to checkout only portions of base, it is easier, until you have experience building base products, to check out the entire base tree. This can be done with the commands:

```
cd <to directory just above the epics created in previous step>
cvs checkout epics/base
cd epics/base
gmake
```

This will checkout and build the entire EPICS base.

Checkout 'extensions'

Other than the EPICS system manager, you will probably never want to checkout the entire set of extension products. Instead you will only want to check out an individual product for development. If the product relies on other products, the other products will be automatically referenced from your site official base and official extensions areas. If you want to change these referenced locations, you can add definitions for EPICS_BASE and/or EPICS_EXTENSIONS to the end of your `<epics>/config/CONFIG` file.

For example, to checkout an extension product, issue the commands:

```
cd <to directory just above epics>
cvs checkout epics/extensions/src/<product>
cd epics/extensions/src/<product>
gmake
```

This will check out and build `<product>`.

3. Using CVS Commands

Committing a New Version

Perform the following steps:

```
cd <epics>/extensions/src/<product>
```

Or

```
cd <epics>/base/src/<product>
```

Modify the files as desired.

```
cvs commit
```

You will be asked to supply a comment for the `cvs` log when you commit modified files.

Other Useful CVS commands

The following are helpful `cvs` commands:

```
cvs add        Adds a new file/directory to the repository
cvs diff       Runs diffs between revisions
cvs help       Prints list of cvs commands
cvs log        Prints out information for files
cvs status     Prints status info on the files
cvs remove     Removes an entry from the repository
cvs update     Brings work tree in sync with repository
cvs commit     Checks files into the respository
```

See the CVS document *Version Management with CVS* by Per Cederquist and the `cvs` man page for details.

4. Using EPICS Makefiles

This section describes how an EPICS developer uses EPICS Makefiles. A basic knowledge of CVS operation is assumed.

Makefile Commands

Each src/xxx directory contains two types of Makefiles, a single high-level Makefile and one or more low-level Makefiles, one for each architecture type to be built (e.g.: Vx or Unix).

While developing code for a particular architecture, a developer can use either of two different methods. The developer may opt to use a higher-level approach and invoke gmake in a src/xxx directory (e.g.: <epics>/base/src/db), or use a lower-level approach and invoke gmake in an architecture-specific object directory (e.g.: <epics>/base/src/db/O.sun4). The higher level Makefile will generate all targets for single or multiple architectures. The low-level Makefile in the O.<ARCH> directory can only generate targets for a single architecture, however it is capable of generating a single object target.

The following make commands are available in EPICS Makefiles:

- **dirs**: Creates the O.<ARCH> directories. Only available in high-level Makefile.
- **depends**: Creates header file dependencies. By default, the compiler used to build is also used to generate dependencies. To use another compiler for this purpose, override DEPENDS_RULE.c and/or DEPENDS_RULE.cc in the CONFIG file.
- **build**: Compile and link non-library objects.
- **install**: Install executables (Unix) or linked objects (VxWorks) in $INSTALL_BIN ($EPICS/base/bin or $EPICS/extensions/bin)
- **clean**: Remove all temporary files from O.<ARCH> directories.
- **all**: Same as install.

The default command is install, i.e.

```
gmake
```

is the same as

```
gmake install
```

The makefiles in the src/XXX directories also support the single architecture "command.arch" syntax. For example, to build and install software for the mv167 architecture, type "gmake install.mv167".

The makefiles in the O.<arch> directories can be used to build a single target. These makefiles do NOT support the single architecture syntax "target.arch" because a single architecture is the default.

For example to build dbAccess.o in $EPICS/base/src/db/O.mv167, execute:

```
cd O.mv167
gmake dbAccess.o
```

Notes:

- To build and install iocCore, invoke "gmake install" in $EPICS/base/src/iocCore.
- To build default.dctsdr and the record header files, type "gmake install" in $EPICS/base/src/ascii.

5. Creating High-level Makefiles

Each src/XXX source directory must contain a high-level Makefile which has the following lines:

```
# Relative pathname to the top of the EPICS tree (required).
EPICS = ../../..
```

```
# Include file containing EPICS definitions (required).
# Include $(EPICS)/config/CONFIG_BASE  - for base products.
# Include $(EPICS)/config/CONFIG_EXTENSTIONS - for extension products.
include $(EPICS)config/CONFIG_BASE

# Include file containing EPICS rules for architectures (required).
include $(EPICS)/config/RULES_ARCHS
```

6. Creating Low-level Makefiles

A low-level `makefile` must exist for each build type supported in a particular source directory. For example, the `db` directory performs both Unix and VxWorks compilation. Therefore it contains a makefiles for both build types: `Makefile.Unix` and `Makefile.Vx`.

Low-level Makefile Structure

Each low-level Makefile is made up of four parts.

- Preamble
- Options section
- Product definition section.
- Rules section

Preamble

The Makefile "preamble" contains relative pathnames to the top of EPICS. Every pathname is specified relative to the `O.<ARCH>` directory, as that is where `gmake` is being invoked.

```
# Relative pathname to the top of the epics tree (required)
EPICS = ../../../..

# Include file that sets the EPICS target architecture (required)
include Target.include

# Include file containing EPICS definitions (required)
#  $(EPICS)/config/CONFIG_BASE- For base products.
#  $(EPICS)/config/CONFIG_EXTENSIONS- For extension products.
include $(EPICS)/config/CONFIG_BASE
```

Options Section

The options section allows options to be specified for the various utilities in the build. The following options are supported:

```
# "Compile type" option, used only for Unix builds. (optional) !  If this option
# is left out, an ANSI compile is chosen by default.
# OLD - Select old-style compilation for this directory.Avoid this when possible.
# STRICT - ANSI compilation with pedantic warning flags.  Use this when possible.
CMPLR = OLD

# C / C++ / loader flags / loader libraries / depending libraries
# VxWorks compiles only support USR_CFLAGS and USR_LDFLAGS.
USR_CFLAGS = -D_NO_PROTO -UvxWorks
USR_CCFLAGS =
USR_LDFLAGS =
USR_LDLIBS = -lDb -lCom -lm
DEPLIBS =

# Lex/Yacc option flags, with typical settings.
YACCOPT = -d -v
LEXOPT = -l
```

Product Definition Section

This section specifies the sources, objects, libraries and library dependencies that will be used to compile targets in this directory.

```
# Source files.  Include targets, but without the ../.
SRCS.c = ../snc_main.c ../parse.c ../phase2.c snc.c snx_lex.c   ...

# Object files.  All object files are created in the current directory.  Do not
# include objects that are parts of libraries on this line.  Leave this
# blank if only libraries are being built in this directory.
OBJS = snc_main.o parse.o phase2.o snc.o snc_lex.o   ...

# Library object files.These are objects that make up the principal library being
# built in this directory.  Leave this out if no libraries are being built.
LIBOBJS = obj1.o obj2.o obj3.o   ...

# Library name. Leave this out if no libraries are to be installed
LIBNAME = libname.a

# Products.Objects specified in $(OBJS) are linked together to create this
# target. Leave this out if no products are to be installed
PROD = prod1 prod2

# Targets to be built but not installed. Leave this out if none are to be built.
TARGETS = target1 target2

# Include files. Leave this out if no include files are to be installed.
INC = include1.h include2.h

# Document files. Leave this out if no document files are to be installed.
DOCS = doc1 doc2 doc3

# Man pages files. Leave this out if no man pages are to be installed.
MAN1 = mana.1 manb.1 manc.1

# Scripts to be installed. Leave this out if no scripts are to be installed.
SCRIPTS =  script1 script2 script3
```

Rules Section

This section specifies the rules that will be used to build the targets in this directory.

```
# Include the default rules file (required).
include $(EPICS)/config/RULES.Unix

#  Add rules for objects not easily built with the default rules file, e.g.:
tsTest: tsSubr.o
    $(LINK.c) -o $@ tsSubr.o -lCom -lm -s

#  Add the "clean::" double colon rule.  This rule should remove all temporary
#  files in the O.<ARCH> directory except those with .o and .a extensions.  These
# files are automatically removed by the default rules file.
clean::
    rm -f snc snc_lex.c snc.c
```

Default rules are provided for `lex` and `yacc` generated C files. As with objects, these temporary files are generated in the `O.<ARCH>` directories. If left alone, `yacc` will create `y.tab.c` and optionally, `y.tab.h`. However, the default rules will rename these files automatically so that `yacc` may be used again in the same directory. For example, if you have a `yacc` file `foo.y`, the default rules will create `foo.c`, and optionally, `foo.h`. This renaming also occurs for `lex` generated files.

Therefore, to use the default rules, all files in the same directory must have different basenames. For example, three files named `dbLoadTemplate.y`, `dbLoadTemplate.l`, and `dbLoadTemplate.c` will confuse the default Make rules. This is not an absolute restriction, as you may override the default rules as you see fit. However, we suggest using the default rules since they result in simpler Makefiles.

C files generated by `lex` are often included directly into C files generated by `yacc`. When this is the case, extra dependency must be created manually in your Makefile, for example:

```
# Extra rule since atdb_lex.c is included in atdb_yacc.c
atdb_yacc.o: atdb_lex.c
```

NOTE: It is recommended that you use the EPICS supplied version of `lex` (`elex`) and `yacc` (`antelope`). To do this, put the following two definitions in your Makefile:

```
LEX = $(ELEX)
YACC = $(EYACC)
```

7. Modifying The CONFIG File

The `epics/config/CONFIG` file may be edited by developers or the EPICS system manager in order to override specific variables in the configuration. Configuration files other than `CONFIG` and the “`SITE`” files are not meant to be modified at all. `CONFIG` provides a convenient place to put all variable overrides.

Say, for example, a developer wishes to turn on compiler warnings that were disabled by the EPICS system manager for the build. The developer would then add the following definitions at the **end** of the `CONFIG` file:

```
VX_WARN=YES
UNIX_WARN=YES
```

Notes: Only the EPICS system manager is allowed to commit this file into the local CVS repository.

8. Developing EPICS Base at a Remote Site

With the help of CVS, EPICS base can easily be developed by remote sites. This is because CVS allows each file to be edited by more than one person at a time, and merges parallel changes when the source code is finally checked in. To develop EPICS base at a remote site:

1. Checkout a copy of EPICS base and config from the CVS repository at the master site. Important: Note time and date of this operation.
   ```
   cvs checkout base config
   ```
2. Create an EPICS release.
   ```
   cd $EPICS/base
   gmake release
   ```

Or obtain an official EPICS release from the EPICS repository site.

3. Move this release to the remote site. Make changes to it, possibly with the aid of a local CVS repository. Note files and directories that have been modified, or use the Unix “`find`” command to perform this operation automatically.
4. Create a new EPICS release and send it back to the master site.
5. Check out the version of EPICS current at the date and time you obtained it.
   ```
   cvs checkout -D "DATE" base config
   ```
   Or checkout the release of EPICS you obtained from the repository site using its tag:
   ```
   cvs checkout -t "EPICS_R3_12" base config
   ```
6. Copy all files that were modified at the remote site onto this version.
7. Update these new files either individually or at the top-level, using
   ```
   "cvs update -A".
   ```

8. Fix conflicts and then commit changes with “`cvs commit`”.

9. Developing New Extensions

Creating a New Extension

To create a new extension within an extensions development tree perform the following steps:

```
cd <epics>/extensions/src
mkdir <newext>
cvs add <newext>
cd <newext>
```

Create and test the `<newext>` files and `Makefiles`. Put the `cvs` keyword `$Log$` someplace in each source file.

```
cvs add <filename>
```

The `add` command is issued for each file to be placed under `cvs` control. Don’t forget the Makefiles.

```
cd <epics>/config
```

NOTE: You may not have file permission for the next step. In this case you will have to ask your EPICS system manager to perform this step.

Add your new extension directory to the directory list in the `CONFIG_EXTENSIONS` file. These directories are used for an extension top-level build.

```
cvs commit CONFIG_EXTENSIONS
```

You will be asked to supply a comment for the `cvs` log when you commit modified files.

Importing an Existing Product

In the source directory where the product currently exists execute:

```
cvs import -m "Imported files" epics/extensions/src/<prod> <prod>_R0 <prod>_R0
```

WARNING: Make sure you have a a clean directory structure before invoking import because it recursively descends all directories from where it is invoked.

If the source files were previously maintained via SCCS, it is best to import a version obtained via the `sccs get` command. If the SCCS keywords such as `%W%` and `%G%` were used then the file will contain a record of the SCCS version that was converted to CVS.

Perform the following steps:

```
cd <to directory just above epics>
cvs checkout epics/extensions/src/<prod>
cd epics/extensions/src/<prod>
gmake
```

Create and test necessary Makefiles and then use the `add` command for each Makefile as follows:

```
cvs add <filename>
```

Create a `.cvsignore` file if desired.

```
cvs add .cvsignore
```

Add the `cvs` keyword `$Log$` to each source file. If a Modification log was manually maintained previously just add the `$Log$` after the last log entry.

```
cvs commit
```

You will be asked to supply a comment for the `cvs` log when you commit modified files.

```
cd <epics>/config
```

Add your new extension directory to the directory list in the `CONFIG_EXTENSIONS` file. These directories are used for an extension top-level build.

```
cvs commit CONFIG_EXTENSIONS
```

You will be asked to supply a comment for the `cvs` log when you commit modified files.

Chapter 7: Site Manager Interface

1. Preliminaries

CVS

Install `cvs` and create a `CVSROOT`.

GNU

Install GNU `make` version 3.71 or later and make sure executing `gnumake` locates this version. Type the following command to make sure you have your path set properly.

```
which gnumake
```

/usr/local/epics

Create this directory and make sure it belongs to you.

Env/Path

Set your Unix environment and path and provide EPICS users with proper Unix environment..

/usr/local/etc/ Cshrc file

The Unix system manager may provide a file that users execute from their .cshrc file to satisfy EPICS (and other) path and environment variable requirements. At APS this file is named:

```
/usr/local/etc/Cshrc.aps
```

The EPICS system manager arranges to have the following two EPICS lines added to this file.

```
# Epics operations
source /usr/local/etc/setup/EPICS
```

If no such Cshrc file exists, the EPICS system manager may arrange to have EPICS users source `/usr/local/etc/setup/EPICS` directly from their .cshrc files.

Note that the EPICS manager should prepare similar files for other shells used at the site.

/usr/local/etc/setup/EPICS

The EPICS system manager provides the `/usr/local/etc/setup/EPICS` file which contains site specific EPICS base and extensions path and environment commands. It must contain at least the following commands (modified for the specific site):

```
#  site-dependant EPICS setup
setenv PSPRINTER lp
setenv EPICS_TS_MIN_WEST 360
setenv EPICS_AR_PORT 7002
source /usr/local/epics/startup/.cshrc
```

/usr/local/epics/startup/.cshrc

The EPICS system manager provides this file which contains the non site specific path and environment commands needed to satisfy EPICS user requirements and EPICS Applications Source/Release Control requirements. The `/usr/local/epics/startup/.cshrc` file contains the following lines:

```
set path = ( ./appSR/tools ./appSR/bin/`arch`  ./base/tools ./base/bin/`arch` $path)
if ( ! $?EPICS_EXTENSIONS ) then
    setenv EPICS_EXTENSIONS /usr/local/epics/extensions
endif
if ( $?EPICS_EXTENSIONS_PVT ) then
    set path = ( $EPICS_EXTENSIONS_PVT $EPICS_EXTENSIONS/bin/`arch` $path)
else
    set path = ( $EPICS_EXTENSIONS $path)
endif
```

NOTE: The Unix system manager may have to provide an "arch" script that returns one of the EPICS supported Unix architectures when executed.

2. How to Build EPICS Base in Six Steps.

1. Make coffee.
2. Obtain a copy of EPICS, and type the following commands:
   ```
   cd epics
   # Use cp or mv to put the EPICS Tar file in this directory.
   # Uncompress if necessary (uncompress RX.XX.X.Tar).
   tar xef RX.XX.X.Tar
   ```
   You will be left with two directories, `config` and `base`.
3. Edit all files in the `config` directory containing the string “`SITE`” that are appropriate to your build. For example, if you are building `sun4` executables and IOC clients, you want to edit `CONFIG_SITE`, `CONFIG_SITE.Unix.sun4`, and `CONFIG_SITE.Vx.68k`. See Chaper 3 Section 3 “Config Sub-Tree” on page 12 for a description of these files.
4. Edit `$EPICS/base/src/misc/epicsEnvParams.h`. This step is optional as these variables may be overridden in an application directory with a `resource.def` file. This file contains these options:
   ```
   /* Minutes west of Greenwich */
   EPICS_TS_MIN_VALUE= “480”;
   /* Address of IOC-log server (a Unix workstation) */
   EPICS_IOC_LOG_VALUE= “128.3.112.85”;
   /* Maximum size (in bytes) of log file */
   EPICS_IOC_FILE_VALUE= “400000”;
   /* Log file name */
   EPICS_IOC_LOG_FILE_TXT= “/home/epics/iocLog.text”;
   ```

NOTE: This step must be changed! Base is not doing this correctly.

5. Select the record/device/driver support you want to use at your site. To do this edit the following files:
    - base/src/ascii
        - dbRecType.ascii
        - devSup.ascii
        - drvSup.ascii
        - choiceRec.ascii
    - base/src/rec/Makefile.Vx
    - base/src/dev/Makefile.Vx
    - base/src/drv/Makefile.Vx

    In each case comment out what you do not want rather than deleting.
6. Build EPICS.
    ```
    cd ../base
    gmake
    ```

NOTES:

- If you receive EPICS as a statically-built binary distribution, steps 3 through 6 may not be necessary !
- If you have already built EPICS or have received a pre-built version, and then decide to change options in `CONFIG_SITE`, you will have to:
    ```
    cd epics/base
    gmake clean uninstall
    ```
    This will ensure EPICS is properly cleaned out for a compile with new options.
- If your site is going to do EPICS development, it is recommended that 1) you place base under `cvs` control and 2) header file dependencies be generated.

3. Other Top-Level Commands for EPICS Base

The top-level makefile of EPICS base can be used for functions other than just building. It can be used for cleaning objects, creating releases and building architectures separately.

Removing EPICS Objects

Temporary objects in the `O.*` directories (and the directories themselves) may be cleaned at the top level by typing:

```
gmake clean
```

All EPICS libraries and executable files within `$EPICS/base/bin` and `$EPICS/base/lib` may be deleted from the top level by typing:

```
gmake uninstall
```

Creating Dependencies

EPICS Makefile dependencies are generated using “`gmake depends`”. This command may be executed at the top-level for all directories, or at a low-level for just that directory:

Top-level:

```
cd $EPICS/base
gmake depends
```

Low-level:

```
cd $EPICS/base/src/db
gmake depends
```

Building a Release

To build an EPICS release,

1. Edit `base/include/version.h` if you need to change the EPICS version number.
2. In `CONFIG_SITE`, make sure the `STATIC_BUILD` option is set to `YES` if you want static UNIX objects. If you change this or any other options, you will have to fully clean EPICS base with “`gmake clean uninstall`”.
3. Type:
   ```
   gmake release
   ```
   This command will recalculate dependencies, build EPICS, and then create a Tar file in the directory above base. This Tar file will not include any binaries. If you want all binaries and libraries to be bundled into the Tar file, type:
   ```
   gmake built_release
   ```
   instead.

Single Architecture Operations

A particular build command may be invoked for a particular architecture. For example, if you only want to build files for the `mv167` architecture and then clean all `sun4` objects, type:

```
gmake install.mv167 clean.sun4
```

Notes:

- A UNIX build must be completed before EPICS can be built for a vxWorks architecture. That is why `HOST_ARCH` is always specified first in `BUILD_ARCHS` within `CONFIG_SITE`.
- The uninstall and release commands do not support single architecture operation.

4. How to Build EPICS Extensions

1. Make certain a built copy of EPICS base is already available at your site.
2. Create an extension development directory structure. Type
   ```
   mkdir epics epics/config epics/extensions epics/extensions/src
   ```
3. Obtain a tar file of the desired EPICS extension(s), and extract the extension(s) into your `epics/extensions/src` directory.
4. Copy the files from your site’s `epics/config` directory into your `epics/config` directory.
5. Check definitions in the config directory files. Make certain that `EPICS_BASE` and/or `EPICS_EXTENSIONS` are appropriate for your build. If the extension you have obtained depends on another extension’s library or header files, make certain that the other extension is available at your site and that `EPICS_EXTENSIONS` is appropriately defined. You can override definitions in your `CONFIG` file.
6. Build the extension. If `XXX` is the extension then type:
   ```
   cd epics/extension/src/XXX
   gmake
   ```
7. If you are will be doing development on the extension, you should 1) place extensions under `cvs` control and 2)generate header file dependencies. To generate dependencies Type:
   ```
   cd epics/extension/src/XXX
   gmake depends
   ```

5. Top-Level Commands for EPICS Extensions

The top-level makefile of EPICS extensions can perform functions for all the extensions defined in your `<epics>/config/CONFIG_EXTENSIONS` file. This makefile will be present if you obtained a tar file of the whole EPICS extensions tree structure. It can be used for building, installing, cleaning objects, and building architectures separately. Usually the top-level makefile is used only by the EPICS System Manager at your site.

Removing Objects

Temporary objects in the `O.*` directories (and the directories themselves) may be cleaned at the top level by typing:

```
gmake cleanAll
```

All EPICS libraries and executable files within `$EPICS/extensions/bin` and `$EPICS/extensions/lib` may be deleted from the top level by typing:

```
gmake uninstall
```

Creating Dependencies

EPICS Makefile dependencies are generated using “`gmake depends`”. This command may be executed after all he extensions have been built at the top-level for all directories, or at a low-level for an individual directory:

Top-level:

```
cd $EPICS/extensions
gmake depends
```

Low-level:

```
cd $EPICS/extensions/src/<prod>
gmake depends
```

Single Architecture Operations

A particular build command may be invoked for a particular architecture. For example, if you only want to build files for the `mv167` architecture and then clean all `sun4` objects, type:

```
gmake install.mv167 clean.sun4
```

Notes:

- The uninstall command does not support single architecture operation.

6. Optional Method for Installing Extension Products

These are instructions for installing a new version of an extension product without removing the previous version. The method has the following features:

1. Products are installed with the name `<prod>xxxx` where `xxxx` is a version number.

2. A soft link points to the latest version.
3. If a user wants to access a previous version he/she can create a soft link to the desired version in the `EPICS_EXTENSIONS_PVT/bin/'arch'` directory.

Most sites will find that all their users will want updated versions of extension products independent of the version of base they are using in their iocs. This can be done as long as the Channel Access protocol doesn't change. Users normally locate extension products via `/usr/local/epics/extensions/bin/'arch'`. They can override the version obtained from here by putting a soft link in their `EPICS_EXTENSIONS_PVT/bin/'arch'` directory. For example if a user finds that a new version doesn't work for his/her application then a soft link to an old version can be created.

In order to support these features, add the following statement at the end of your `CONFIG` file:

```
INSTALL_PRODUCT = $(EPICS_BASE_TOOLS)/InstallVersion
```

When a "`gmake install`" is made in one of the `extension/src` directories the result is the following:

- Assume the product is named `<prod>`. When the previous version was created it was named:
  `<prod>xxxx`
  and a soft link was also created with the command:
  `ln -s <prod>xxxx <prod>`
- When the new version is created the previous version is kept and the new version named
  `<prod>yyyy`
  where yyyy is equal to xxxx + 1

  The soft link for <prod> is changed to reference the new version.

You will want to set up operational procedures so that users are properly notified when new versions are installed. These procedures should include information stating the previous and current version numbers.

Chapter 8: Build Internals

1. Configuration Files

It is not often necessary to understand the “internals” of EPICS configuration files if you merely want to develop or build EPICS code. However, an understanding is required if you are porting to a new architecture, for example.

This section describes the files contained within `$EPICS/config`, with the exception of the “`SITE`” configuration files that were described earlier in Chapter 3 Section 3 “Config Sub-Tree” on page 12.

The low-level extension and base Makefiles directly include what are called the “EPICS top-level include files.” They define the directories that make will traverse through when base and extensions are being built. Targets are built in the order that they appear in the DIRS variables.

- **CONFIG_EXTENSIONS**: Included in extension Makefiles.
- **CONFIG_BASE**: Included in low-level base Makefiles.

`CONFIG_EXTENSIONS` and `CONFIG_BASE` both include the `CONFIG` include file which in turn includes the following files (in this order):

- **CONFIG_SITE**: Top-level site definitions.
- **CONFIG_COMMON**: EPICS base definitions.
- **CONFIG_ARCH.$(ARCH)**: e.g.: `CONFIG_ARCH.mv162`
  Defines `BUILD_TYPE`, `ARCH_CLASS` and architecture dependent C flags.
- **CONFIG_SITE.$(BUILD_TYPE).$(ARCH_CLASS):**
  e.g.: `CONFIG_SITE.Unix.sun4`
  Architecture-class / build-type dependent site definitions. These files are described in Chapter 3 Section 3 “Config Sub-Tree” on page 12.
- **CONFIG.$(BUILD_TYPE).$(ARCH_CLASS)**: e.g.: `CONFIG.hkv2f.68k`
  Compiler definitions.

The CONFIG file may be edited either by individual developers or the EPICS system manager. It provides a centralized, convenient place to override configuration variables.

The BUILD_TYPE variable is set to either "Vx" or "Unix," depending on the architecture being built. Sun workstations (sun4) have a BUILD_TYPE of "Unix", whereas the mv167 has a build type of "Vx". The ARCH_CLASS variable provides a logical way of organizing compiler flags that are reused by different architectures. For instance, mv167, hkv2f, and mv147 targets share much of the same configuration. Once defined in CONFIG_ARCH.*, BUILD_TYPE and ARCH_CLASS are used to control what files are subsequently included in CONFIG.

The CONFIG.*.* files contain definitions that are shared by every architecture in a specific architecture class. It creates the compile and link lines based on variable settings in CONFIG_SITE and variable overrides in CONFIG.

A RULES.* file is included near the bottom of every low-level Makefile. RULES.* files contain default rules that simplify dependency definitions. Rules are provided for building objects, installing objects, and generating source files with lex and yacc.

RULES.Vx and RULES.Unix in turn include the .DEPENDS file in the O.ARCH directories. .DEPENDS consists of automatically generated header file dependencies that are created when "gmake depends" is invoked. When the O.ARCH directories are first created, an empty .DEPENDS file is created so that initially make will work. It is the responsibility of the EPICS developer to manually run "gmake depends".

2. Adding a New Architecture

To add support for an architecture in the EPICS build, follow these steps: ($ARCH is the architecture being added):

1. Create CONFIG_ARCH.$ARCH. Set BUILD_TYPE, ARCH_CLASS, and the architecture
   a. Set BUILD_TYPE. If this is an IOC architecture, set to Vx. If this is a host architecture, set this to Unix.
   b. Set ARCH_CLASS. If this architecture is 680X0 based, set to 68k. If it is a host architecture, set this variable to $ARCH.
   c. Set architecture dependent variables:
   ```
   #  Architecture dependent C flags.
   ARCH_DEP_CFLAGS=
   #  Architecture dependent C++ flags.  Probably doesn't make sense.
   ARCH_DEP_CCFLAGS=
   #  Architecture dependent LD flags.
   ARCH_DEP_LDFLAGS=
   #  Architecture dependent libraries.  These include
   $  compatibility libraries, for example.
   ARCH_DEP_LDLIBS=
   ```
2. Create CONFIG.$BUILD_TYPE.$ARCH_CLASS if it doesn't already exist.
   a. Copy either CONFIG.Unix.sun4 or CONFIG.Vx.68k as a beginning.
   b. Add support for local compilers. Add choices for both ANSI C and standard compilers specific to that platform. In addition, GNU CC should always be supported in addition to the vendor supplied compilers. For instance, sun4 host architectures have the following choices for ANSI compilation:
      - ACC - Sun's unbundled compiler product (acc).
      - GCC - GNU CC (gcc -traditional).

      And the following choices for standard C compilation:
      - SUNCC - Sun's traditional compiler (cc).
      - ACC - Sun's unbundled compiler product (acc -Xs).

- GCC - GNU CC (`gcc -ansi`).

(Note: These Compiler choices are provided to the local system manager in the `CONFIG_SITE` file).

c. Add support for warning generation, dependency generation, and static build flags. To properly generate these header file dependencies, a macro must be added to the file CONFIG.<BUILD_TYPE>.<ARCH> for `DEPENDS_RULE.c` (C sources) and `DEPENDS_RULE.cc` (c++ sources). For example the file `CONFIG.Vx.68k` contains the statements:

```
DEPENDS_RULE.c = -$(COMPILE.c) -M $(SRCS.c) >> .DEPENDS
DEPENDS_RULE.cc = @echo no DEPENDS_RULE.cc defined in CONFIG.Vx.68k
```

Note that a double arrow is used to direct output to `.DEPENDS`

d. If this is a UNIX build type add support for the compilation types (`OLD`, `ANSI`, and `STRICT`).

3. Create `CONFIG_SITE.$BUILD_TYPE.$ARCH_CLASS` if it doesn't already exist. Copying either `CONFIG_SITE.Unix.sun4` or `CONFIG_SITE.Vx.68k` is probably sufficient.

Chapter 9: Tools Manual Pages

1. Introduction

EPICS shellscripts are found in the `base/tools` subdirectory of the EPICS distribution. Of these scripts, only gmake, getrel and makesdr should be invoked directly.

2. gmake

This script executes `gnumake` with the flags:

--no-print-directory --no-keep-going .

3. getrel

```
getrel EPICS_PATH [arch list]
```

This script sets up EPICS application directories by creating soft links to EPICS and vxWorks releases.

Example:

```
getrel ../../r3.11.4 mv167 hkv2f
```

4. makesdr

```
makesdr
```

`makesdr` creates a custom database template (`default.dctsdr`) for an application directory. This command is optional because a default template will be used if this command is not invoked.

5. MakeRelease

```
MakeRelease [-b]
```

`MakeRelease` creates a Tar image of an EPICS release. The `-b` option means binaries in the `bin/*` and `lib/*` directories are included in the release. However, the `default.dctsdr` and `default.sdrSum` files are still included if `-b` is not used.

6. MakeDirs

```
MakeDirs
```

Creates temporary object directories under an EPICS release for a particular architecture.

7. GetVar

```
GetVar VAR_NAME [ARCH]
```

Gets make variable information from within a shell script.

Common use:

```
GetVar HOST_ARCH
```

This will retrieve the host architecture variable's value from `CONFIG_SITE`. Use this instead of `/bin/arch` for determining the host architecture from within shellscripts, as it is more portable.

8. CheckArch

```
CheckArch ARCH
```

Determines if configuration files exist for this architecture.

9. FindEpics

```
FindEpics
```

Determines where EPICS is located from within shell script. Many shell scripts invoke this shellscript first.

10. Clean

```
Clean [ARCH]
```

Removes temporary objects from the `O.ARCH` directories under `src`. If `Clean` is called without any arguments all `O.ARCH` directories are removed.

11. makeStatTbl/blderrSymTbl

```
makeStatTbl
blderrSymTbl
```

Used for creating and building the error symbol table.

12. SetOpVersion

```
SetOpVersion
```

Used to select specific executable versions of EPICS extension programs that will be used on future command invocations.