Created:August 18, 1994
Modified:January 16, 1997

Gemini Controls Group Report

Software Programming Standards

Jim Wright, Bret Goodrich, Steve Wampler

SPE-C-G009/05

This report details Gemini Project Software Programming Standards.

1.0 Introduction

1.1 Purpose

The purpose of this document is to establish software programming standards and practices to be used in the development and maintenance of the Gemini software. Those identified as standards by the use of the words "must/shall" will be audited for compliance. Those identified as practices by the use of the words "should/may" are included only as good programming conventions and techniques or as objectives to be employed. These standard and practices achieve the following objectives:

- Insure that design, coding, and testing are accomplished in a manner which complies with the Gemini Software Management Plan.
- Establish terminology which allows for production of a well-structured modular software design that is traceable to and will satisfy system requirements.
- Establish appropriate uniformity of format and program annotation for ease of maintainability.
- Establish coding standards which minimize interface problems and maximize the effectiveness of the programs generated.

As there are several systems commercially available (CodeCheck, *et cetera*) that will do auditing of coding formats the GPCS will be able to perform compliance tests on the code generated by the WPR during the acceptance test phase of each work package.

1.2 Scope

The standards and practices in this document apply to software developed by Gemini and any parties explicitly contracted to create Gemini packages and modules. This document specifically addresses the coding standards to be employed.

Other software development issues are dealt with in other Gemini documents, specifically:

- Standards to be employed during the design phase of the software development [1].
- Standards to be employed during the testing phases of the software development [4].
- Documentation standards for software related documents [2].

1.3 Previous Standards

All code written prior to the release of this document shall conform to either version 03 (previous) or version 04 (current) of the Software Programming Standards. If code must be modified to bring it in line with one of these standards, you are **strongly** encouraged to follow the most recent version of the SPS. Code which was written prior to this version of the SPS (version 04, December 1996) may abide by the standards set forth in the previous version of this document. However, this version of the SPS does not significantly differ from previous versions. It clarifies a number of issues, fixes a large number of typographical and formatting errors in the document, and even relaxes the standards in a number of areas.

All code written after the release of this SPS shall abide by the standards set forth herein.

1.4 Applicable Documents

The following documents are official Gemini documents, referred to in the text, even if not all of them exist yet. These documents will be provided to all developers of Gemini software both to provide useful background and to provide the environment under which the Gemini software will be generated.

Software specifications and management

[1] (SRS) *Gemini Software Requirements Specification*, [SPE-C-G0014]

[2] (SMP) *Gemini Software and Controls Management Plan*

[3] (SCCP) *Gemini Software Configuration Control Plan*, [SPE-C-G0011]

[4] (STP) *Gemini Software Test Plan*

[5] (SDD) *Gemini Software Design Description*, [SPE-C-0037]

[6] *Gemini Acronym Glossary*, Ruth Kneale, [PG-S-G0008]

[7] *Large-Scale C++ Software Design*, John Lakos, Addison Wesley

[8] *Software Engineering for ESO's VLT Project*, G. Filippi

[9] *Very Large Telescope Programme - Software Management Plan*, G. Filippi, [VLT-PLA-ESO-00000-0006]

[10] *VLT Software Release 1 (Common Software) Overview and Installation Manual*, G.Filippi, [VLT-MAN-ESO-17200-0642]

Tcl/Tk information

[11] *Tcl and the Tk Toolkit*, John K. Ousterhout, University of California Berkeley

[12] *Tcl/Tk Engineering Manual*, **ftp://ftp.sunlabs.com/pub/tcl/engManual.tar.Z**, John K. Ousterhout, Sun Microsystems Inc.

[13] **http://www.sco.com/Technology/tcl/Tcl.html**, Tcl WWW Info

[14] **http://www.sunlabs.com/research/tcl/**, Tcl/Tk Project at Sun Microsystems

[15] **ftp://ftp.smli.com/pub/tcl/**, Main Tcl/Tk distribution

[16] **http://www.NeoSoft.com/tcl/**, Contributed Tcl/Tk sources

EPICS information

[17] **http://www.atdiv.lanl.gov/aot8/epics/epicshm.htm**, Los Alamos National Lab site for EPICS information

[18] **http://www.aps.anl.gov/asd/controls/epics/EpicsDocumentation/WWW-Pages/EpicsFrames.html**, Argonne National Lab site for EPICS information

[19] **http://www.cebaf.gov/accel/documents/epics_doc.html**, Continuous Electron Beam Accelerator Facility site for EPICS information

[20] **http://csg.lbl.gov/CSG.html**, Lawrence Berkeley National Lab site for EPICS information

[21] *ICD 13 -Standard Controller*, Bret Goodrich and Andrew Johnson

[22] *ICD 1.9/3.1 - The OCS/CICS Interface*, Steven Beard, ROE

[23] *Keck EPICS Programming Manual*, Keck Software Document 82, W. F. Lupton

[24] *EPICS Input Output Controller Record Reference Manual*, Janet B. Anderson and Martin R. Kraimer, Argonne National Laboratory, December 1994

[25] CapFast Electronic Circuit Design CAE User's Guide, Phase Three Logic, Inc., 1991

[26] *EPICS Input / Output Controller (IOC) Application Developer's Guide*, Martin R. Kraimer, Argonne National Laboratory, November 1994

[27] **http://www.atdiv.lanl.gov/aot8/epics/OPI/opintro-1.html**, *EDD/DM USER'S MANUAL*, Version 2.3, Philip Stanley, Los Alamos National Laboratory, November, 1996

[28] *State Notation Language and Sequencer's User's Guide*, Version 1.9, Andy Kozubal, Los Alamos National Laboratory

[29] **http://www.atdiv.lanl.gov/aot8/epics/capfast/capcomref.html**, CapFast Command Reference, Los Alamos National Laboratory

Supporting tools information

[30] *Version Management with CVS*, Per Cederqvist

[31] **http://www.cyclic.com/**, CVS information

[32] **http://www.iac.honeywell.com/Pub/Tech/CM/**, Configuration Management FAQ

[33] *RCS - A System for Version Control*, Walter Tichy, Purdue University

[34] **ftp://cs.purdue.edu/pub/RCS/**, main RCS distribution

[35] *GNU Make - A Program for Directing Recompilation*, Richard M. Stallman and Roland McGrath

[36] **ftp://prep.ai.mit.edu/pub/gnu/**, main GNU distribution

[37] **ftp://ftp.keck.hawaii.edu/pub/wlupton**, main WFLMAN distribution

Some of these documents were not yet ready at the time this specification was written. However, they belong to the currently planned set of documents for Gemini.

1.5 Glossary

CICS - Core Instrument Control System work package

OCC - Optical Components Controller - that part of the instrument control system that controls the non-detector portions of the instrument

OIWFS - On-instrument WFS - the Wavefront Sensor located in the instrument

TCS - Telescope Control System

WFS - Wavefront sensor

1.6 Revision History

- August 1993, version 01, original revision
- August 1994, version 02, convert to Framemaker
- January 1996, version 03, update
- December 1996, version 04, revise and extend most of the document
- January 1997, version 05, release to change control

Part I UNIX Development and VME Support

This part of the Software Programming Standards provides information on programming language selection, coding practices and style, module layout and formatting, and other related issues. Since the code development for EPICS is actually done under a UNIX cross-compilation environment, most of this part will also apply to EPICS developers.

2.0 Language Selection

2.1 Compiled Languages

ANSI C is the language that shall be used for all new work. The traditional K&R C is not found to be a standard in practice. No language extensions (`asm` directives, *et cetera*) are allowed. Specifically, the GNU `gcc` ANSI C compiler is required for both native UNIX work and for VxWorks cross-development. No errors or warnings of any type are allowed during the compilation process using the required `-ansi`, `-pedantic`, and `-Wall` options.

2.2 Object Oriented Alternative

C++ is an acceptable alternative to ANSI C. Its concept of object-oriented programming may benefit some later projects. Specifically, the GNU `g++` C++ compiler is required for both native and cross-development object-oriented applications. Any use of C++ should be explicitly justified by the WPR.

2.3 Interpreted Languages

Tcl/Tk (Tool Command Language) is the language that shall be used for all interpreted applications. This includes incremental Tcl/Tk, also known as [incr tcl] and [incr tk]. See Section 7.0, “Tcl/Tk Style Guide,” on page 21 for more information on Tcl/Tk.

2.4 Imaging Language Alternative

PVWAVE is an acceptable alternative to Tcl when it can be shown that the task benefits from existing support in PVWAVE (image analysis applications, image rendering, *et cetera*). Any use of PVWAVE should be explicitly justified by the WPR

2.5 Emergency Use Alternatives

In the event that C is not available on the target machine, use the following rules to help select a language:

- First choice is a modern, structured language with object oriented concepts.
- Second choice is a traditional, less structured language such as FORTRAN.
- Avoid use of assembler if at all possible. If at all possible avoid in-line use of assembler in high-level languages. Most high level languages will be significantly easier and more reliable during the development cycle. Use assembler only in small

subroutines which require ultimate speed or hardware accessibility. Such subroutines shall be well commented, especially in regard to parameter passing.

- Use of BASIC, FORTH and system manufacturer proprietary languages is unacceptable.
- Use of the C shell or derivatives for shell scripts is unacceptable. All shell scripts should use the Bourne shell. If a need is documented by the WPR, use of a POSIX compatible shell (such as the Korn shell) is acceptable. However, no non-POSIX features or extensions may be used.

Any use of languages outside of ANSI C, C++, Tcl/Tk and PVWAVE in permanent code shall be approved by the Gemini Controls Group prior to their use in the project.

2.6 Pre-existing Code

When working on or porting existing code, individual decisions should be made on whether old sources should be kept in the original language or converted to the project standard.

Some points to consider are:

- The existence of a good compiler for the original language may allow keeping the original code.
- If the original code cannot be easily compiled on the new machine, consider conversion before extensive modifications in the original language.
- A good calling interface between new C programs and subroutines in another language may allow keeping the original code.
- If the subroutine calling and parameter passing is difficult, or requires changing original code, consider conversion to C.

2.7 Preprocessors

The use of programming tools that act as sophisticated preprocessors to C shall be checked in advance to ensure appropriate support for the tool exists within the Gemini Project. The following GNU tools are recommended by the Gemini Project Office.

TABLE 1. GNU Tools

GNU Tool	Function
gawk	Pattern scanning and processing language
flex	Lexical analysis program generator
bison	Yet another compiler-compiler
indent	C source code formatter
make	Maintain, update, and regenerate related programs and files

3.0 Coding Practices

The following practices and standards apply not only to C programs under UNIX, but also to C programs for EPICS, Tcl/Tk and PVWAVE programs. The book *Large Scale C++ Software Design* [7] provides the guidelines for C++ developers which should be followed for Gemini software.

3.1 Modular Coding

Functions should be designed so that they contain 200 lines or fewer of executable C statements. It would be permissible to exceed this in the case of large `switch` statements which should not be broken.

Functions should typically perform a single logical operation.

Functions should have a single entry point and a single exit point.

3.2 Function Declarations

All functions externally available should be declared in a separate include file.

All functions in the module should be declared at the beginning of the file. <u>Functions that are internal to a source code module shall be declared `static`.</u>

<u>Function declarations shall be of the full ANSI prototype format.</u>

Naming conventions are defined in Section 6.0, "Naming Conventions," on page 20.

3.3 Data Declarations

External data references should be explicitly declared. Names of external data items should reflect their location.

Implicit dependence on data type size in a particular C implementation should be avoided. For example. when a 16-bit quantity is desired, make a `typedef` of `INT16` to what is actually a 16-bit integer on the particular machine. This will be placed in a Gemini standard include file which will keep machine dependencies together and well noted. This include file will be created and maintained by the GSCG.

Naming conventions have been established for different data types and usages, especially for external shared data. See Section 6.0, "Naming Conventions," on page 20.

<u>Data types shall not be mixed across function boundaries</u>. Type casting should be used where necessary. Use of ANSI parameter checking in the compiler will help find such errors.

3.4 Include Files

Use of include files is encouraged to clearly identify all constant information shared by different modules in a project and to avoid duplication of information in files. Use file name only or a name relative to some defined directory. `#include` statements shall not reference an absolute path.

Put all “address” type constants, such as hardware addresses, in include files where only one change will be necessary, rather than requiring editing many files.

Define constants for frequently used items and use them in comparisons in preference to numeric values. Such constants should be in include files, and well documented.

Use standard include files when they are available. Use the standard definitions in these files to help standardize programs.

Include files shall be written so duplicate inclusions of the same file does not cause any errors.

No storage allocation or initialization shall occur within any include file.

History information is given by the CVS/RCS `$Log$` keyword.

The `*INDENT-OFF*` and `*INDENT-ON*` pair protects the formatting of the comments from modification by the “indent” program (see Section 11.2, “Indent,” on page 36). The `*+` and `*-` pair are used by the off-line utility WFLMAN to extract these comments and produce documentation (see Section 11.1, “WFLMAN,” on page 36).

The include file template appears below.

```
/*
 * Copyright 1996 Association of Universities for Research In Astronomy, Inc.
 * See the file COPYRIGHT for more details.
 *
 *   FILENAME:
 *   Put file name here
 *
 *   PURPOSE:
 *   Short description of what the include file does
 *
 *INDENT-OFF*
 * $Log$
 *INDENT-ON*
 */

#ifndef MODULE_NAME
#define MODULE_NAME

/* put definitions here */

#endif /* MODULE_NAME */
```

3.5 Hardware Access

All access to hardware should be isolated to libraries. If a library does not provide the needed functionality, fix the library instead of working around it.

Only one process or task should be allowed direct access to any raw hardware.

Libraries or functions should be developed to perform high level hardware actions, or to replace repeated, confusing low-level library calls.

3.6 Errors

All error conditions should be considered and flagged for the calling routine's attention. The existence of an error condition should be made available to the error logging and error recovery systems. The types of errors defined by the Gemini software are listed in the SRS [1]. The standard format of error messages, which includes identification of the responsible module, is covered in the error logger description in the SRS. The formal treatment of error and fault conditions can be found in the SDD [5].

3.7 GOTO

`GOTO` is considered a poor programming practice and should not be used when a normal programming construct could be used instead. Use it only for non-recoverable errors and exception handling. <u>`GOTO` shall only be used to jump downward in the code, typically to an error handler below the normal return.</u>

Non-local `GOTO`s like `setjmp` and `longjmp` should not be used.

3.8 Style Considerations

Programmers who work on the code of others should respect the style of the original programmer and not leave any “scars”. Follow the style of the original programmer, even if it differs from your own or from the standard style. The only exception is to redo the whole module. <u>If you redo the whole module, you must adhere to the Gemini standards.</u>

Avoid programming “tricks”. Any code which is not immediately apparent should be well commented.

Spelling and grammatical errors will not be tolerated.

3.9 Comments

Comments should be written to help some anonymous stranger understand the code years later. Insufficiently commented code is unacceptable.

Comments should be kept up to date and should help explain the overall concept of the algorithm, not exactly detail what the code statements already show.

3.10 Computer Error Checking

Code shall compile without any warnings or errors under the `gcc` compiler using the `-ansi -pedantic -Wall` options.

All code should be written to ANSI standards, especially as they involve prototype and function declarations. When using an ANSI compliant compiler all checking of parameter passing should be turned on.

3.11 Numeric Constants

Use `#define` to assign meaningful names for all constants. The names of constants should use only upper case letters, digits and underscore.

Use of numeric constants or "magic" numbers in code or declarations should be avoided except for obvious uses of 0, 1, *et cetera*.

3.12 Structure Definitions

Always define structures and unions by a `typedef` statement. Never use a structure or union definition to declare a variable directly. All `typedef` statements should be placed in include files.

3.13 Argument Passing

The passing of structures and unions by value is supported by the ANSI C standard and is allowed within the Gemini Project.

3.14 Function Status Return

Routines that return good/bad type values should return predefined `PASS` or `FAIL` values that are consistent over an entire project. `PASS` and `FAIL` will be defined in a Gemini supplied include file.

Specific error information should be returned by other means:

- A program could use a global error variable. However, do not subvert the normal operation of the standard UNIX `errno` global variable for your own purposes.
- A standard error handling function can be written to display all error messages. This topic touches on the design of error logging and error recovery systems which is covered in the SDD [5].

3.15 Templates

The use of the GCSG-supplied templates for include files and function/procedures is required. These templates are found in Section 4.2, "Module Header," on page 15 and Section 4.3.1, "Function Header," on page 16 of this document.

A template file should be copied to each new source file you create and then edited to make a new source. A template should also be added to the top of existing files so they conform to these standards.

4.0 Module Layout

A module is any unit of code that resides in a single source file. Thus a module may be a UNIX tool, a library of routines, or an application task. The conventions in this section mainly define the standards for the module header which must come at the beginning of every source module.

4.1 Version Control

Version control shall be provided by the CVS configuration management system. For more information, see Section 14.0, "Source Control," on page 39.

4.2 Module Header

The module header consists of the four mandatory blocks described below. The header comes at the top of the file. The items should be in the order presented.

Version control identifier tag: This shall have the standard CVS form.

File name: Every module should include the name of its source. Use the file name only or a name relative to some defined directory, rather than giving an absolute path.

Function name(s): The name of each function in the module must be listed along with a single line comment describing what it does.

History: A history section should list every modification of the file, in a "who, when, what" format. This history is for the entire file and is provided by the CVS `$Log$` keyword.

If this file is used as a template for a state notation program (i.e. one ending with the file extension .st or .stpp), the line declaring `rcsid` **must** be moved to after the snc `program` statement and then escaped using the `%%` construct. Failure to do this will cause the snc compiler to fail.

```
static char rcsid[]=”$Id$”;
/*
 * Copyright 1996 Association of Universities for Research In Astronomy, Inc.
 * See the file COPYRIGHT for more details.
 *
 * FILENAME
 * Put file name here
 *
 * FUNCTION NAME(S)
 * The name of each function in the module goes here along with a single
 * line comment on what it does e.g.
 * function1 - initializes the other functions, must be called first.
 *
```

```
*INDENT-OFF*
* $Log$
*INDENT-ON*
*/
```

4.3 Function Layout

A function is a single C function. A module may contain a single function, a `main()` and supporting functions, or a library of functions. All functions are basically treated the same and each should have the same documentation. If a module contains a `main()` that should come first, with all supporting functions below it.

4.3.1 Function Header

The function header is the comment block which shall be included before every function. Each item in the function header should have the title and all items are mandatory. If there are no entries for a particular item, still include the title, and enter “none” for that item.

Each header should be done as a single large comment. The function header for the first function must be in a separate comment from the module header. This header is intended to be all another programmer needs in order to use the function. These items will be stripped out and used as documentation for the function. The items should be in the following order and follow the following formatting.[1]

Function name: The name of the function.

Invocation: An example C statement showing how the function may be called. See the WFLMAN documentation for information on how to control the formatting of the generated text (bulleted lists, enumerated lists, verbatim text, etc.)

Parameters: The formal parameters which are passed in to or back from the function. The first token on the line should be indicate if this parameter is input, output, or both input and output. Enclose the symbol within parentheses. Next give the parameter’s name. After this give the parameter’s type, enclosed in parentheses. Finally describe the parameter more fully giving usage and real world units the parameter represents. Any limits on the input parameters should be documented here. The information for each parameter may extend on to the following lines.

Function value: The value and type of the C function return. This should not be confused with the parameters passed back by the function.

Purpose: Description of what the function does. If necessary, describe the algorithm used. If similar functions exist, describe why this one is unique.

Description: A short description of the algorithm used by the function.

1. Already existing code which contains the following entries but perhaps in a different order need not be modified just to place entries in the “correct” order.

External variables: A list of all global variables which are referenced or may be affected by this function. This includes globals external to the module as well as variables declared globally within the module. Also include static variables local to the function which may affect subsequent calls to the function.

Prior requirements: Document everything which must be done prior to calling this function. Include other functions which must first be called, such as initialization functions.

Deficiencies: List known problems, bugs or design limitations for this function.

Note that you may include other sections of documentation in this header, and you are encouraged to do so if the complexity of the code warrants it. Expressions below which appear in parentheses should actually preserve the parentheses. Expressions in square brackets should be replaced with the appropriate information. The comment after the “**Parameters:**” line should be left in place verbatim.

Also note that these header examples are formatted for C code, but these templates must be used for all code. For example, Tcl/Tk code would use `#+` to start comments, `#` to continue comments, and `#-` to end comments.

```
/*
 *+
 * FUNCTION NAME:
 * Place function name here
 *
 * INVOCATION:
 * How to call the function or procedure
 *
 * PARAMETERS: (">" input, "!" modified, "<" output)
 * (CODE) name ([C type]) [parameter description]
 *
 * FUNCTION VALUE:
 * ([C type]) [return description]
 *
 * PURPOSE:
 * Purpose of this routine
 * Try to make the first line a succinct description
 * as it will be used as a section title. Also do not
 * terminate the first line with a period.
 *
 * DESCRIPTION:
 * A short description of the algorithm used by the function
 *
 * EXTERNAL VARIABLES:
 * Any external variables used in this function
 *
 * PRIOR REQUIREMENTS:
 * Operations that must be performed before calling this function
 *
 * DEFICIENCIES:
 * Any known problems with the function
 *-
 */
```

4.3.2 Function Body

The function body is the minimum you could get away with, if there weren't any standards like this. The parts of a typical function are listed below in the order they should appear in the function definition. If an item is not needed then just omit that section. It is not necessary to label each section, but you should separate sections with a blank line.

Include files: Put all `#include` statements first. Any `#defines` that are required for proper operation of the include files must be put before them. Comments are encouraged.

Constant definitions: These are all the `#defines` used to declare constants used in the function.

Macro definitions: These are all the `#defines` used to declare real macros as opposed to constants. The distinction between this item and the one above may blur at times, so use your judgment. If a constant or macro definition is used non-locally it should be placed in an include file.

Function definitions: This is a definition for any function declared static to this module. If ANSI C is in use, this should be the full function prototype. External function definitions should be obtained from `#include` files. Externally visible functions in this module should be defined in this module's include file.

Global variables: Declare all data items which are to be defined outside the function definition. They may be external data items or globals local to the current module. If local to the current module, the variable may also be initialized here.

Function declarations: This is the actual declaration of the function which includes all the function's formal parameter declarations. Use the ANSI format.

Start function block: This is the `{` used to start the function block. It should be at the left margin.

Local variables: All of the variables local to the block should be declared at the top of the block. The suggested order is to declare `char` first, then `int`, working up to longer data types. Each variable should be declared on a separate line with a comment to identify its usage. Any initialization of local variables should be made quite apparent.

Body of function: See Section 5.0, "Source Code Format," on page 20.

Normal return: This is the exit if no errors occur. Every function should have a `return()` statement at the bottom. The `main()` function should have an `exit()` call at the end. It is considered bad practice to "return" out of the middle of a function or to "fall through the bottom" of a function.

Error return: If the function uses `GOTOs` to handle exceptions (allowed but not recommended), they should jump to a location below the normal return. It should handle whatever needs to be done to generate error messages or set up error codes, and then `return()` with the appropriate error code.

End function block: This is the `}` used to close the function block. It should be aligned with the `{` used to start the block.

```
/*
 * function name()
 */
#include <stdio.h>

/* constants */

/* functions */

/* global data */

main (int argc, char *argv[])
{
    /* code */
}
```

5.0 Source Code Format

This is the graphic layout of the source code which gives it a characteristic appearance. There are many “standards” in this area, given in many C language books, and seen in many programmer's work.

The GNU source code formatting utility `indent` will be used to maintain a common appearance for the C language programs created by Gemini work packages.

6.0 Naming Conventions

6.1 File Names

Unless the compiler or operating system requires special names, make use of common extensions for file names.

TABLE 2. Filename extensions

File Type	Name
C language source	*.c
C++ language source	*.C
Object module	*.o
C language header files	*.h
UNIX library (archive) files	*.a
Tcl source files	*.tcl, *.itcl
Tk source files	*.tk, *.itk
PV-Wave procedures	*.pro
CapFast EDIF documents	*.sch
EPICS SNL source	*.st, *.stpp

TABLE 2. Filename extensions

File Type	Name
EPICS Database short report form	*.sr
EPICS Database link report	*.link
EPICS Database one-line summary	*.one

6.2 Programming Construct Names

The following guidelines shall be used when naming Gemini project modules, routines, variables, constants, macros, types, and structure and union members.

- Names shall be made meaningful and readable. Remember, this code must be maintained by anonymous strangers at some distant time in the future.
- Names of routines, variables, typedefs and structure and union members shall be written with upper and lower case and with no underlines. Each word except the first is capitalized.

  aVariableName
- All constants, and macros shall be all upper case with underlines separating the words in the name.

  A_CONSTANT_VALUE
- Every module has a short prefix (2-5 characters). This prefix shall be attached to the modules name and to all externally available routines, variables, constants, macros, and typedefs. Local names are not required to follow this convention.

  modules: ccdLib.c

  subroutines: ccdInitialize()

  variables: ccdExposureTime

  constants: CCD_TYPE
- Pointer variable names shall have a prefix “p” attached for each level of indirection.

  ccdImage *pCcdFrame;
- Each shared memory area must have a structure name. The name must be meaningful, and should also be the name of the shared memory access structure. Using “sh” as the first part of the name will further identify it as shared memory. For example `shCcdControl` might be the structure name for a memory block.

7.0 Tcl/Tk Style Guide

In general Tcl code written for the Gemini project shall conform to the standards defined for C software in terms of comments, naming conventions, styles, and layout. A good reference for Tcl/Tk development is the book *Tcl/Tk Engineering Manual* [12]. There are, however, a few Tcl/Tk specific observations to be noted for Gemini developers.

7.1 Namespace Pollution

To avoid namespace pollution incremental Tcl (also known as [incr tcl] or as itcl) is recommended.

7.2 Built-in Features of Tcl

7.2.1 Dynamic Loading

All Tcl/Tk addons must use the dynamic loading capability of Tcl. The old method of generating a new version of the `wish` executable is not acceptable.

7.2.2 -textvariable

Use of Tcl's `-textvariable` feature is very valuable. Use it for

- Accessing the current value in an **entry** widget.
- Setting the value of a **label** widget that needs to change as a result of user input or as a status monitor.

7.2.3 trace

Tcl's `trace` facility can be very helpful if you need some action to be performed when a particular variable changes. For example, here is an example of maintaining a formatted version of a variable, displaying 2 significant digits.

```
trace variable prf w prfWritten
# This procedure is called when the prf variable has changed.
# We set the name of the menu button to show what was selected.
#
proc prfWritten {varName dummy op} {
    global prf
    global fmtPrf
    set fmtPrf [format "8.2f" $prf]
}
```

7.2.4 Loading

Use dynamic loading for all compiled extensions. To load scripts, use the autoload facility.

```
lappend auto_path /usr/local/lib/tcl_local $env(RDS_TCL_SCRIPTS)/lib $src_dir
```

7.3 Widgets

7.3.1 Meta Widgets

Use of incremental Tcl meta widgets is encouraged.

7.3.2 Widget Groups

When creating a set of widgets, write a short procedure to create each group of widgets. The procedure should take the top level name (typically a frame) as an argument, and should return that widget's name to the caller. This makes it easier to move groups of widgets to a different place in the widget hierarchy, and tends to avoid having lots of hard-wired pathnames.

7.3.3 Widget References

Minimize using global variables or long widget pathnames. For example, you want to avoid writing code that relies on knowing that another widget is called `top.next.lower.buttons.ok`. It is far better to pass the widget name to a function. However, if you do need to use a widget pathname in another function, it is better to set a global variable with the widget name, rather than hardwiring the widget name in that function.

7.3.4 Re-usable Functions

Build your functions to be re-usable. For example, pass “callback” procedures as arguments, rather than hardwiring their names, or using global variables. E.g., if you were writing a function that presented a dialog to accept a user supplied string, pass the function an argument of a procedure to call when the “OK” button is pressed, rather than setting a global variable.

7.3.5 Visual Style

Use `-relief sunken` to make entry boxes obvious.

7.4 Version Dependencies

It is not acceptable that software depends on the TCL_LIBRARY or TK_LIBRARY environment variables. It is strongly advised that users do not set these variables. If a module requires a specific version of Tcl/Tk, then it can be directly invoked using the following as the first line of your script.

```
#!/usr/local/bin/wish4.0 -f
# Copyright 1996 Association of Universities for Research In Astronomy, Inc.
# See the file COPYRIGHT for more details.
```

Or you can write a shell wrapper to “do the right thing” and then invoke the script.

```
#!/bin/sh
# Copyright 1996 Association of Universities for Research In Astronomy, Inc.
# See the file COPYRIGHT for more details.
#
# Shell wrapper for tkinfo.tcl. Idea is to allow PATH search for wish binary.
# The "$@" syntax is important for getting proper quoting. This must be
# a Bourne script: there is no way to get proper quoting and pass through "~"
# in the csh.
#
cd /usr/local/lib/tkinfo
exec wish3.3 -f /usr/local/lib/tkinfo/tkinfo.tcl "$@"
```

Part II VME Specific Development

This part addresses concerns only relevant to EPICS developers. This includes issues such as naming conventions, coding style and visual style.

8.0 EPICS Development

The environment for running VME-based applications is EPICS. No WPR shall build any VME-based application that does not use EPICS as its primary environment.

8.1 Database Creation

The EPICS database creation tool is CapFast. The use of either GDCT or DCT to create EPICS databases is disallowed. In some cases it may be useful to hand-craft a database directly into an ASCII short report. This should be limited to a very few exceptional cases where the number of identical records is very large.

If a few database record and field naming conventions are followed, then it is possible to generate a uniform interface to the OCS command layer.

Because of the arbitrarily complex nature of an EPICS database it is generally not possible to define a one-to-one mapping between interface hardware module descriptions (cardtype/cardno/signal) and instrument control functions. This is because the mapping is dependent upon the interaction between the driver/device support routines and the database itself. The EPICS database, with its use of forward processing, input links, output links, and fanouts, allows the programmer to create database records that by virtue of their link structures implement device control concepts at a higher level of abstraction than what is expressed at the device driver level.

For example, an IR Detector Controller may have a database record that expresses the concept of readout sequencing. Each of these readout actions in turn may have many actual hardware actions to perform.

8.2 Name Conventions

EPICS records may have names up to 29 characters long; this size is a limitation within EPICS. Records are named in a hierarchical manner in the following form:

```
<prefix>:<name1>:<name2>:...:<record>.<fieldname>
```

where `<prefix>:` is the name of the top-level CapFast diagram, `<name1>:`, `<name2>:`, *et cetera* are the names of the hierarchical objects defined using CapFast, `<record>` is the bottom level record name, and `<fieldname>` is the name of a field within the record. The `<fieldname>`, if omitted, defaults to `VAL`. The entire record name, including the "`:`" separators, must be no longer than 29 characters.

8.2.1 System Prefixes

In the Gemini Control System, each system has a unique prefix which, when applied correctly, guarantees no name conflicts between systems. The prefixes for the principle systems and TCS subsystems are defined in Table 3 on page 25.

TABLE 3. System Naming Conventions

System	Prefix
Observatory Control System	`ocs:`
Data Handling System	`dhs:`
Core Instrument Control System	`cics:`
Telescope Control System	`tcs:`
Acquisition and Guiding	`ag:`
Adaptive Optics	`ao:`
Cassegrain Rotator	`cr:`
Enclosure	`ec:`
Primary Mirror	`m1:`
Secondary Mirror	`m2:`
Mount	`mc:`
Interlocks	`is:`

The purpose of system prefixes is to insure uniqueness between all Gemini Systems. For systems which are not listed, the developers, in agreement with the Gemini Project, should choose a unique, four-letter abbreviation for their instrument. This abbreviation will be declared in an instrument's Software Design Document.

8.2.2 Instruments and Detector Controllers

The list of existing instruments is given in Table 4 on page 26. Instruments consist of the Components Controller and the Detector Controller. All records comprising the Components Controller will have a name beginning with the defined instrument prefix and an additional prefix of "`cc:`". For instance, the NIRS Components Controller would have the prefix "`nirs:cc:`".

All Detector Controllers are considered to be subsystems of an instrument. During development of a Detector Controller an instrument prefix or other unique name may be used. However, the secondary prefix must be "`dc:`". For instance, the near-IR Detector Controller during development would have the prefix "`naac:dc:`", but once installed as the detector for the NIRI and NIRS instrument it would have either the prefix "`niri:dc:`" or "`nirs:dc:`". Once a detector has been integrated with an instrument the database names must be adjusted. This step is performed during the commissioning of an instrument/detector pair.

TABLE 4. Instruments and Detector Controllers

Instrument	Detector Controller
High Resolution Wavefront Sensors	`hrwfs:`
Gemini Multi-Object Spectrograph	`gmos:`
High-Resolution Optical Spectrograph	`hros:`
Michelle	`mch:`
Mid-Infrared Imager	`miri:`
Near-Infrared Imager	`niri:`
Near-Infrared Spectograph	`nirs:`
Near-Infrared Detector Controller	`naac:`

8.2.3 Wavefront Sensors

For the most part, wavefront sensors follow the same conventions as science detectors. Each WFS has both a detector and a mechanism that is used for positioning the detector. These are separated in the EPICS name space by different prefixes.

The HRWFS is treated as a separate instrument and is not logically a subsystem of the A&G. The Detector Controller will have the prefix "`hrwfs:dc:`" and the Components Controller will have the prefix "`hrwfs:cc:`".

Peripheral wavefront sensors are a subsystem of the A&G and will have the prefix "`ag:`".

The on-instrument wavefront sensor detectors are logically part of the A&G unit and will be fully prefixed: for example, "`ag:wfs:gmos:`".

The on-instrument wavefront sensor mechanisms are logically a part of the instrument and will be prefixed with that instrument's name. For example, the GMOS WFS mechanism is "`gmos:wfs:`".

8.3 EPICS Engineering Screens

Engineering screens will be created using **dm**, an EPICS extensions tool from LANL [27]. An alternative method of creating engineering screens is the use of Tcl/Tk as provided by the Gemini EPICS release.

8.3.1 dm Displays

Displays used for engineering purposes that are developed using **dm** should have a similar look and feel about them. Custom graphics may be used where it increases comprehension or simplifies use of the display.

1. Every screen must reference a central color template to minimize the number of colors allocated. The Gemini color template file "colors.adl", provided in the EPICS release, should be used.

2. Colors must be used for the purposes stated in Table 5 on page 27. The standard colors defined within this file must not be modified. New colors should only be added where absolutely necessary and only after seeking permission from the Gemini project. They must be added at the end of the existing list. The standard color rules defined in “colors.adl” should be used where possible. Other rules may be added for specific purposes but Gemini must be notified of these rules and they must be made as general as possible. Useful general rules will be incorporated into future releases of “colors.adl”.
3. The template file “template.adl” should be used as a starting point for **dm** screens. Some standard widgets may be obtained from this template, but the **dm** screen is not restricted to these. Additional **dm** widgets may be used where necessary, especially if this makes the screen easier to understand. Every screen must have an EXIT button, as shown in “template.adl”. Custom widgets built using Tcl/Tk may be used where “dm” lacks the required functionality.
4. Every system should have a top-level screen on which the overall health and status of that system can be seen. If possible, a summary of the health and status of all of the underlying mechanisms or subsystems should also appear on this screen.
5. Inputs and outputs should be clearly differentiated on all screens.
6. It must be possible to test the system-to-OCS interface using one or more of the system’s **dm** screens. This is required for acceptance testing.
7. All **dm** screens should be saved to ASCII “.adl” files to guard against changes to **dm** and **edd**. When a system is released every “.dl” file must also be saved as an “.adl” file.

8.3.2 dm Colors

The Gemini project has adopted baseline templates for color tables and rules used in **dm** displays. These templates are provided with the Gemini EPICS releases. The following table describes the standard Gemini colors and their intended use with **dm** displays.

TABLE 5. Gemini EPICS color table

Use	Index	Color Name	RGB Value/Flags			
Default Foreground	1	White	255	255	255	
Default Background	2	Black	0	0	0	
Major Alarm/Error	3	Red	255	0	0	flashing
Minor Alarm/Error	4	Yellow	255	255	0	flashing
No Alarm/Good	5	Green	0	255	0	
Grey Foreground	6	Grey75	191	191	191	
Input Background	7	GeminiBrown	193	156	128	
Output Background	8	Grey54	138	138	138	
Exit Button Foreground	9	Black	0	0	0	
Exit Button Background	10	Tan3	205	133	64	
Title Foreground	11	Black	0	0	0	
Subtitle Foreground	12	Blue	0	0	255	
Heading&Box Foreground	13	NavyBlue	0	0	128	
GeneralDisplay B/G	14	PeachPuff	255	218	185	

TABLE 5. Gemini EPICS color table

Use	Index	Color Name	RGB Value/Flags			
BUSY	15	White	255	255	255	flashing
PAUSED	16	LightSlateBlue	132	112	255	
IDLE	17	White	255	255	255	
Default Color rule F/G	18	Cyan	0	255	255	
Default Color rule B/G	19	Black	0	0	0	
Default Graphics F/G	20	Black	0	0	0	
Default Graphics B/G	21	Grey75	191	191	191	
Generic Black	22	Black	0	0	0	
Generic White	23	White	255	255	255	
Generic Red	24	Red	255	0	0	
Generic Green	25	Green	0	255	0	
Generic Blue	26	Blue	0	0	255	
Generic Yellow	27	Yellow	255	255	0	
Generic Magenta	28	Magenta	255	0	255	
Generic Cyan	29	Cyan	0	255	255	

8.3.3 dm Color Rules

The following color rules are defined as Gemini standards in the color template. The intended purpose of usage is also stated. These rules may not be modified. Additional rules may also be created by developers, but they should be made as general as possible and submitted to the Gemini Project for consideration.

alarm. “alarm” is the default color rule, which **edd** supplies for you. When an object on a **dm** screen has its color modified by this rule, the color will turn flashing red when the specified EPICS record goes into a MAJOR alarm state or flashing yellow when the record goes into a MINOR alarm state. The color is green when there is no alarm and cyan if the alarm state is invalid.

health. This rule is normally connected to a SIR record containing a description of the health of a system. The object modified by this rule is displayed green if the health record contains “GOOD”, flashing yellow if the health record contains “WARNING”, flashing red if the health record contains “BAD” or cyan if the contents of the health record could not be recognized.

limit. This rule turns an object flashing red if the value of an EPICS record drifts out of the range specified.

flashifbusy. This rule is normally connected to a CAR record. It makes an object flash between white and grey when the CAR record is “BUSY”. The object is a steady white when the CAR record is “IDLE”, becomes light blue if the CAR record is “PAUSED” or flashing red if the CAR record becomes “ERR”. The color changes to blue if the CAR record becomes “UNAVAILABLE”.

greyifbusy. This rule is similar to “flashifbusy”. It is normally connected to a CAR record. Like “flashifbusy”, the display will turn light blue if the CAR record is “PAUSED” or flashing red if the CAR record becomes “ERR”. The main difference is that with this rule the display turns grey if the CAR record is “BUSY” and white when the CAR record is “IDLE”.

greyifnotbusy. This rule is almost identical to “greyifbusy” except the display changes to white when the CAR record is “BUSY” and grey when the CAR record is “IDLE”.

healthbusy. This is a complex rule that looks simultaneously at two EPICS records. One of the records should be a health record and the other should be a CAR record. The color of the display gives a summary of the status of a mechanism based on these two records. The value stored in the health record takes priority, and the CAR record is only looked at when health contains “GOOD”. Here is a summary of the rule:

- If health is “BAD”, color is flashing red.
- If health is “WARNING”, color is flashing yellow.
- If health is undefined, color is cyan.
- If health is “GOOD” then
  - If CAR is “IDLE”, color is white.
  - If CAR is “BUSY”, color is flashing white/grey.
  - If CAR is “PAUSED”, color is light blue.
  - If CAR is “UNAVAILABLE”, color is blue.
  - If CAR is “ERR”, color is flashing red.
  - If CAR is undefined, color is cyan.
- End if

8.4 Initialization of EPICS Databases

Most EPICS records may be initialized within the database with constants attached through hardware inputs. However, there are occasions where the initialization depends upon the application of the database; for example, the TCS must initialize the Gemini North database with different values (latitude, longitude, *et cetera*) than the Gemini South database. These variables are best initialized using the **pvload** utility.

8.5 Programming in EPICS

There are two extremes found in EPICS programming:

- Coding *everything* directly into the EPICS database
- Coding *as little as possible* into the EPICS database

In practice, neither of these is likely to be acceptable or desirable. Some common sense is required in determining the appropriate point at which straight C code is a better choice than connections between records in an EPICS database. For example, it would definitely be inappropriate to recast SLALIB astrometric library routines into a database form. Instead, they can be encapsulated in EPICS subroutine records and fit into a EPICS database seamlessly and easily, without modifying the C code in the routines.

There are three methods to select from when doing EPICS development:

- EPICS database code [24],
- C code encapsulated in record, device, or driver support [26],
- State-Notation Language (SNL) [28].

Which vehicle is most appropriate is a judgement call for each task being implemented. The choice should be based upon the specific requirements and design of each task and developers should be prepared to defend their choices. However, the order of the above methods should be used to attack any problem. First, write as much of a database as possible using existing EPICS records and devices. When this method cannot solve the problem, look into writing a new record. In its simplest form, this could merely be the use of a subroutine or similar record. For problems which deal directly with external interfaces such as hardware, device support may be the only answer. Finally, use SNL for sequencing state changes that are too complex or cumbersome for databases alone to handle. The SNL code definitely should not contain any parameters that are not in the database. Some sequencing also can be done within the database using the *wait* record, for example.

8.5.1 EPICS database code

The arguments in favor of concentrating most of the structure in the database are:

- Code reuse. The main function of an EPICS software engineer is to write the record support, device support, driver support, and client software that is required for the application and not available elsewhere. Thus a major benefit of EPICS is software reuse. If you analyze your problem and attempt to generalize it, then you are likely to come up with a plug-in component that is of general use. SNL and subroutine records fail in this major area.
- The many useful client side tools that already exist. These are made possible by the client/server architecture and the simple name/value relationship that the database represents. The database, at heart, is a name/value system. It is best to analyze where it is deficient and fix it by adding suitable record types and client software.
- The fact that the database drawing is both documentation and truly represents the database structure. This avoids the problem of documenting procedural code or SNL with diagrams that can date quickly and are often difficult to update since they were initially constructed with specialized tools. You are also forced to standardize your representation, so in the long term more people have an ability to understand the system.

However, attempting to put all information, regardless of its complexity, into a database comprised solely of the standard EPICS records is a futile job. Databases cannot implement every feature a complex task may require. When this point is reached, the developer must look towards writing C code for new EPICS records or SNL code for the sequencer.

8.5.2 Writing EPICS Support Routines

One of the features of EPICS is the ability to extend its functionality by writing additional code for record, device or driver support. Information on how to write these support routines is given in the EPICS literature; however, determining when and where to write new routines is a matter of some importance.

A record is the basic element of any EPICS database. There are many types of records currently available in EPICS, but sometimes none of them, or any combination of them, can satisfy the requirements of the task. If you find that you can use a subroutine record and write C code to do the job, then you may wish to consider writing a custom record to replace the subroutine record. If you use this subroutine record in more than one place, a new record is recommended.

New records should be generalized as much as possible to support reuse by other work packages. Writing a record with a too specific function will remove any possibility of sharing it. The Gemini Project should be informed of new records as they are being written. New records should be written with device support hooks, even if the envisioned use of the record does not include device support.

Device support for EPICS records should be written whenever a database will be accessing hardware or virtual devices. Like record support, the implementation should be generalized where possible to support code reuse. Most EPICS records have hooks for device support, and these records should be used first. Device support should be written for as broad a range of records as possible; for example, a numeric value from a device should be accessible from a *longin*, *ai*, or *stringin* record.

Driver support should be written for any new hardware created by a work package. This support should be written for the VxWorks operating system, and not necessarily for EPICS. The goal of driver support is to provide drivers which operate in a VxWorks environment without the use of EPICS. However, keep in mind while writing the driver support that EPICS device support will be the ‘glue’ between the hardware and the database. Keeping driver support to the standard EPICS device support calls of **init**, **init_record**, **get_ioint_info** and **read** or **write** will simplify the device support layer.

8.5.3 State Notation Language Support

State Notation Language is one of the most powerful run-time tools in EPICS. With SNL a very complex set of states can be defined to describe how a database should transition. Because of this power, conditions which cannot be adequately described in a database or record can be monitored and handled with ease. SNL should be used in these situations; for example, when the state of operation of a database (booting, configuring, running) needs to be selected, or when a global error condition needs to change the way a database operates. The ability to disable record processing, and thus prevent certain sections of the database from executing, is also a powerful capability of the sequencer.

SNL code should not be used to directly control hardware or devices, to execute subroutines, or to handle links between records that could be done by the database. Doing these will remove any visibility into the database for the EPICS tool set. SNL code must not contain any parameters that are not maintained in the EPICS database. (The only exception being counters and other trivial variables concerned with the control of the SNL itself).

The use of the variable substitution facility of the sequencer is encouraged, as this allows several instances of the same SNL code to be run at the same time with different variable settings, allowing SNL to be reused. If the CapFast macro substitution facilities are used variables with the same purpose should be given the same names.

8.6 EPICS Lock-sets

EPICS records are often connected by flows that place the records on both ends of the flows into the same ‘lock-set’. When processing occurs within any member of the lock-set, no new inputs are accepted into the lock-set. If an input record is part of a large lock-set, then the implication is that further attempts to use that input record are not possible until *all processing in the lock-set is completed.* Consequently, care must be taken when programming with EPICS databases to check that the lock-sets created in the database are not unreasonable.

As a general rule of thumb, any link from one database record that produces a flow triggering processing to occur in the other record places both records into the same lock-set. So, for example, any forward links from one record to another put both of them into the same lock-set. A less obvious case is that a “process-passive” link with one record taking input from a second record puts both into the same lock-set.

In general, one must design in specific breaks to prevent the growth of a lock-set. This may be done by using “non-process-passive” input fields, events (not recommended as a general solution since the number of events is restricted), judicious use of state machines, *et cetera*.

Please note that lock-sets are a useful method in EPICS database design to prevent input values from changing during critical sections of code. Their use is not discouraged or minimized, but rather should be carefully thought through during design and development.

9.0 UAE Development

All EPICS development will be within the Universal Application Environment (UAE) [23] developed by William Lupton (Keck Observatory) and Nick Rees (JACH). The standard UAE templates for make and resource files are distributed by the Gemini Project Office.

9.1 Setting up UAE

UAE provides a means of developing and releasing application code in a fashion similar to the way EPICS code is developed. The UAE environment can be split into two different areas: the development and the release directories. In the most trivial application, the two directories are identical. For more sophisticated applications the directories should be split to provide a released version independent of the development environment.

UAE also works well with the CVS source code management system; developers are strongly encouraged to keep all EPICS work under CVS version control to facilitate the distribution of prototypes and beta-releases to the Gemini Project.

A typical execution of the *applSetup* command would describe the install directory, the UAE subdirectories to include or exclude, the subdirectories to compile, and dependencies to other applications.

```
applSetup -install <release_pathname> \
            -include config \
            -targets mv167 \
            -subdir_file Makefile.subdirs \
            -depends <other_application_pathname>
```

9.2 Locating Files

9.2.1 Start-up files

The VxWorks start-up file should be located in **startup/startup.vws**. This file should reference the file **startup/local.vws** for all site-specific startup information such as routers and security.

All start-up files will obey the stylistic standards defined in Part I of this document.

9.2.2 Source code

The UAE templates provide two types of source code development directories, ‘src’ and ‘sys’. The difference between them is how the makefile compiles the final object file. In the case of the ‘src’ directory there is one object file for every source file. For the ‘sys’ directory there is one object file for all the source files; the object file is named after the directory, i.e., **sys.o**.

Developers should use either of these directories to hold their source code. The Makefile in both directories should be edited to add the relevant source files in the **SRCS.c** rule. Either directory structure may be copied into new directories at the top of the UAE application.

9.2.3 Include Files

All include files should be placed in the application **include** directory.

9.2.4 Parameter Files

All parameter files such as lookup tables, pvload files, *et cetera*, which are used by an EPICS application should be located in the application **data** directory.

9.2.5 CapFast Schematics and Symbols

All schematic and symbol files generated by CapFast should be located in the application **capfast** directory. If the application has created new records and CapFast symbols for these records, a new **edb.def** file must also be placed in the directory and the **Makefile.Unix** file variable E2SR_SYSFLAGS must be changed.

```
E2SR_SYSFLAGS = -atec -d ../edb.def
```

The **edb.def** file needs to have values for the new records added to it. The base version of this file can be found at **${EPICS}/extensions/src/edif/lib/edb.def**.

9.2.6 Record and device support

The ASCII files associated with records and device support should be place into one of either **ascii/cat_ascii** or **ascii/replace_ascii**, depending upon whether the record or device support adds to or replaces existing records. The file **dbRecType.ascii** lists the record support information and the file **devSup.ascii** lists the device support information.

9.3 Distributing a UAE application

Releases of Gemini applications such as the CICS or On-instrument Wavefront Sensor packages are distributed as compressed tar files of the UAE environment. Object files and site-specific files are omitted from the release.

9.3.1 Creating a release

A release file can be generated from the UAE application by running **gmake release** from the top directory of the application. Instructions on how to install the release, including the appropriate *applSetup* arguments, should be written in a top level **README** file. The top level **Makefile** should be modified to include a **gmake setup** rule.

9.3.2 Unpacking a release

To unpack a distributed release:

1. Create a directory to unpack the release into.
   % mkdir *rel*; cd *rel*
2. Transfer the release file into this directory.
3. Unpack the release
   % gunzip -c *release.tar.gz* | tar xf -
4. Follow the instructions in the **README** file to create the UAE environment.
   gmake setup*(if the developer has provided this feature).*
5. Compile the application
   gmake

10.0 CapFast Development

CapFast [25] is perhaps the best tool for EPICS database construction, largely because its ability to hierarchically present an EPICS database permits information hiding and reduces the visual complexity of the database. This section presents some recommended practices when constructing EPICS databases with CapFast [29].

<u>All CapFast drawings shall use a size C border around all symbols and wires in the drawing.</u> No elements of a drawing should be outside of this border. The symbol for the size C border can be found in the EPICS parts list. Information about the drawing (file

name, designer, date, project, comments, revisions, *et cetera*) should be placed in the appropriate information blocks in the lower right and upper right corners.

All symbols should be given names which are indicative of their purpose; default naming should not be used. All symbol names should be made visible. Symbols should have their hierarchal prefix given either by creating the *PV* property or using the *name* property.

Whenever a symbol's default properties are modified or appended, the altered property should be made visible and displayed just below the symbol.

The use of bus wires in a drawing is encouraged where the need is evident. A bus wire has the advantages of consuming less space and simplifying wiring. However, wires selected to be grouped into a bus should have some strong association; for example, the apply record's OUTA-OUTH fields. Wires connected to the bus must be labelled.

10.1 Hierarchal Capabilities

EPICS database designers using CapFast are ***strongly*** encouraged to take full advantage of the hierarchical capabilities of CapFast to encapsulate and clarify logical functional units within the database. This is analogous to the use of appropriately-sized modules when programming in traditional, text oriented languages. As a general rule-of-thumb, a single view of a database should not contain more than ten functional blocks and fewer if there is a large number of connections between the blocks.

Free text should be associated with the views to explain unusual or critical operational steps.

Portions of EPICS databases that perform similar functions should be consistently represented. Identical functional units should be described using a single CapFast hierarchical object that is then instantiated as often as needed (much as a function in C is used to avoid duplicating common code).

Part III Development Support

This part addresses procedures and tools to be used in support of software development.

11.0 Other Tools

11.1 WFLMAN

The WFLMAN program [37] is used to automatically scan source code and generate documentation in a variety of formats. The comment-block standards defined in this document are designed to work well with WFLMAN.

11.2 Indent

The indent program is use to format source code according to a standard set of rules. This package is now maintained by the GNU people. You can retrieve indent via anonymous FTP from prep.ai.mit.edu in the directory /pub/gnu. The options used for indent for all Gemini project work are

```
indent -kr -cdb -sc
```

which generates the traditional Kernighan & Ritchie style with the comment delimiters on blank lines and with an asterisk on the left side of all comments. For example:

```
/*
 * This is a single line comment transformed by indent -kr -cdb -sc.
 */
```

The expansion of the above options for `indent` version 1.9.1 is given here.

TABLE 6. Indent flag settings for Gemini use

Option	Description
-nbad	Do not force blank lines after declarations.
-bap	Force blank lines after procedure bodies.
-nbbb	Do not blank lines after block comments.
-nbc	Do not force newline after comma in declaration.
-br	Put braces on line with 'if', *et cetera*, and structure declarations.
-c33	Put comments to the right of code in column 33.
-cd33	Put comments to the right of declaration in column 33.
-cdb	Put comment delimiters on blank lines (**not K&R**).
-ce	Cuddle else and preceding '}'.
-ci4	Continuation indent of 4 spaces.
-cli0	Case label indent of 4 spaces.
-cp33	Put comments to the right of '#else' and '#endif' statements in column 33.
-cs	Put a space after the cast operator.
-d0	Set indentation of comments not to the right of code to 0 spaces.
-di1	Put variables in column 1.

TABLE 6. Indent flag settings for Gemini use

Option	Description
-nfc1	Do not format comments in the first column as normal.
-nfca	Do not format any comments.
-i4	Set indentation level to 4 spaces.
-ip0	Indent parameter types in old-style function declarations by 0 spaces.
-l75	Set maximum line length to 75.
-lp	Line up continued lines at parentheses.
-npcs	Do not put space after the function in function calls.
-npsl	Put the type of a procedure on the same line as its name.
-sc	Put the '*' character at the left of comments (**not K&R**).
-nsob	Do not swallow optional blank lines.
-nss	Do not force a space before the semicolon after certain statements.
-ts8	Set tab size to 8 spaces.

12.0 Installation Directory Structure

At this time the directory structure for the installation tree is still being designed. This document will be updated to incorporate that design. A few aspects of the system directory tree can be addressed now.

12.1 Developed Code

TBD

12.2 Tools

The GCSG tries to make use of existing third-party tools when possible. These include such packages as the following.

- EPICS
- VxWorks
- GNU Tools
- Tcl/Tk

12.2.1 EPICS

The EPICS development environment will be built and installed, as much as possible, according to the existing EPICS conventions with Gemini-specific recommendations found in ICD 13 [21]. The source tree will be as distributed by the EPICS consortium with minimal Gemini additions.

12.2.2 VxWorks

The VxWorks development environment will be built and installed according to the recommendations of Wind River Systems with Gemini-specific recommendations found in ICD 13 [21].

12.2.3 GNU Tools

All GNU tools will be built and installed according to usual procedures. Executables will be installed in `/usr/local/bin`. Documentation will be installed in `/usr/local/man`, `/usr/local/doc` or `/usr/local/info` as appropriate. Libraries and run-time support files will be installed in `/usr/local/lib`. In general, by defining the base directory as `/usr/local`, all GNU tools will be correctly installed.

12.2.4 Tcl/Tk

Tcl and Tk will be built and installed according to usual procedures. With recent versions, this means that multiple versions may co-exist at the same time. For older versions of Tcl/Tk, some modifications to the distribution are necessary so that it will install in a multiple-version-friendly manner. (For example, changing the distribution to use `/usr/local/lib/tk3.3` rather than just `/usr/local/lib/tk`.) It is strongly recommended that all Tcl/Tk code be maintained so it is compatible with the most recent Tcl/Tk version approved for Gemini use.

12.2.5 Other Tools

In general, all other tools should be installed in the `/usr/local` tree as appropriate. Some packages might best be installed in their own hierarchy, for example `/usr/local/vni` for PVWAVE from Visual Numerics Inc.

12.3 Observing Home Directories

TBD

12.4 User Home Directories

The home directory layout for users at their home institutions is at their discretion. At Gemini, all users will have a single home directory. For example, `user`'s home directory would be `/home/user`.

13.0 Makefile Standard

All Makefiles used within the Gemini project shall be compatible with the GNU make facility [35].

One of the most important areas addressed by the standard Makefile is the installation of files into the destination directory tree. This issue has not yet been finalized. This should not prevent design and development of work packages. This document will be updated when this information is available.

13.0.1 Gemini Make Include

There shall be a Gemini Project standard make include file which will be provided by GPCS. It will define a number of rules, macros and defines which will simplify and standardize the work package Makefiles.

TBD

13.0.2 Work Package Makefiles

Each application or subsystem will have its own Makefile. It will `include` the standard include file described above and will make use of its features. An example work package Makefile is shown below.

```
TBD
```

14.0 Source Control

14.1 Gemini Standards

The Gemini standard for configuration management and source control is CVS [30]. Developers are encouraged to use this at their home institutions. The GSCG will maintain a master repository for released versions of all software. Intermediate versions of packages may be included as well at the discretion of the GSCG.

At this time, no standards have been set for directory layout or other constraints on work packages for inclusion in the master repository. However, EPICS development should adhere to the established EPICS idiom as used by UAE (see Section 9.2, “Locating Files,” on page 33).

14.2 Gemini Master Source Distribution

A master source distribution will be derived from the most current released versions of all packages in the Gemini master repository. For some packages such as EPICS, this distribution will define the Gemini baseline and any deviation from this must follow established change control procedures. For other packages, it is anticipated that having other group’s code available will speed the climb up the learning curve and promote interaction between WPRs.

At this time, procedures and timelines for the master source distribution have not been set. It is anticipated that this will entail a custom CDROM mastered by the GSCG and distributed to the WPR sites.