EUROPEAN SOUTHERN OBSERVATORY

VERY LARGE TELESCOPE

VLT Software

–

Programming Standards

Doc. No. VLT-PRO-ESO-10000-0228

Issue 1.0

Date 10/03/93

	Name	Date	Signature
Prepared	G.Filippi	10/03/93	
Approved	G.Raffi		
Released	M.Tarenghi		

This page was intentionally left blank

Change Record

Issue/Rev.	Date	Section/Parag. affected	Reason/Initiation/Documents/Remarks
1.0	10/03/93	All	First issue

This page was intentionally left blank

Contents

1 INTRODUCTION

1.1 Purpose

This document lists the programming standards to be used in the design and development of the VLT Software.

The main aim in defining standards is to create a frame where each development can be done independently but with a common style in order to make future maintenance easier. Maintainability is an essential requirement in a project of the dimensions of the VLT Software, carried on by different development groups, both internal and external to ESO, on a long development and operation period and in a critical logistic situation.

Both commercially available and specially developed tools are foreseen to allow the checking of standards by the developer, during the implementation phase, and to be used as part of the acceptance of software, during the test phase. While it is essential to automate checking of the application of standards as much as possible, where an automatic tool is not available, manual inspection will be used.

The document is organized in sections, each one dealing with a specific topic. The present edition copes with the following arguments:

- Organization of a Software module (section 2)
- Naming conventions (section 3)
- C language (section 4)
- Script files (section 5)
- Make (section 6)
- UNIX (section 7)
- VxWorks (section 8)
- Format and media for deliverable software (section A)

In the document, *must* and *shall* are used to indicate mandatory practices, *should* and *may* are used, for recommendations and guidelines, respectively.

The intended audience of this document are software designers and developers and it is assumed that the reader has a good knowledge of UNIX and C-language. Although not required, familiarity with the VLT Project, in general, and with the classification of software, as described in [3] and [4] is suggested.

1.2 Scope

This document covers the general organization of software and the use of the basic programming environment: UNIX and the C-language.

It is not within the scope of the Programming Standards to specify design rules. As a reminder, there follows a list of some of the standardization design principles:

– the use of services and utilities provided by the CCS (including rules for command naming, error messages, etc.).

- the use of services and utilities provided by the LCU Common SW (including rules for command naming, error messages, etc.).
- the common model of software organization for some typical application (instruments, drivers, etc.).
- the use of standardized software as building blocks for applications (sequencer, CCD, etc.).

For more information, please refer to the appropriate documentation, as indicated by the design specification of each application.

This document applies to all software belonging to the **VLT Software** [1] that will be developed by ESO or by external contractors at both workstation (WS) and Local Control Unit (LCU) levels.

For **External Software**, compatibility with the present document is not required, but a general evaluation of the maintainability of such software should be part of the Design Review. In this case, the Programming Standards can be used as a reference term to measure the deviation of the examined software with respect to the majority of the VLT Software.

Being part of the external software, **embedded software** for transputers, microcontrollers, etc., is not included in the scope of the document. On a project specific basis, this document can be used as a reference to produce analogous standards for such environments.

All the aspects concerning SW Configuration Control, and use of SCCS tool, are covered by the Software Configuration Control Plan [5] (currently in preparation). Nevertheless some parts of the Programminf Standards are influenced by Configuration Control matter and the necessary concepts are anticipated here.

1.2.1 Statement of Applicability

The standards in the following section are applicable to all software defined in the scope of this document. At the beginning of each section, possible restrictions in the application of the standards are stated.

Any additional limitation to those described by this document shall be defined during the design phase and approved during the design review(s). The limitation/extension shall be documented by explicit reference to the sections of the present document that are not applicable.

For software developed under contract, any limitation/extension shall be subject to contractual negotiation. Additional/alternative standard(s) can be used under the condition of being explicitly approved by ESO. The additional/alternative standards shall be clearly identified by referring to existing document(s), either private or in the public domain.

Therefore, in the Technical Proposal (for contracted software only), in the Functional Specification and in the Detailed Design of any specially developed software, one of the following shall appear:

1. "The provisions in the VLT-PRO-ESO-10000-0238 VLT Programming Standard, *issue date (if any)* and in
   reference — scope of applicability — possible restrictions
   reference — scope of applicability — possible restrictions
   .....
   shall be applied in the design and development of the software."
2. "The provisions in the VLT-PRO-ESO-10000-0238 VLT Programming Standard, *issue date* as at section(s):

[1] More about software classification can be found in [3] 2.3.1 and [4]) 1.7.

(section reference) (section title)
(section reference) (section title)
.....
(if any) and in
reference — scope of applicability — possible restrictions
reference — scope of applicability — possible restrictions
.....
shall be applied in the design and development of the software."

3. "All the provisions in the VLT-PRO-ESO-10000-0238 VLT Programming Standard, *issue date* with the exception of the following section(s):
(section reference) (section title)
(section reference) (section title)
.....
(if any) and in
reference — scope of applicability — possible restrictions
reference — scope of applicability — possible restrictions
.....
shall be applied in the design and development of the software"

4. "The provisions in the following Standard(s):
reference — scope of applicability — possible restrictions
reference — scope of applicability — possible restrictions
.....
shall be applied in the design and development of the software."

1.3 Applicable Documents

The following documents of the exact issue shown form a part of this document to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this document, the contents of this document shall be considered as a superseding requirement.

[1] ANSI Standard X3.159-1989 — Programming Language C,

[2] IEEE Std. 1003.1-1990 — Portable Operating Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]

1.4 Reference Documents

The following documents contain additional information and are referenced in the text.

[3] VLT-PLA-ESO-00000-0006, 2.0 21/05/92 — VLT Software Management Plan

[4] VLT-SPE-ESO-10000-0011, 2.0 30/09/92 — VLT Software Requirements Specification

[5] VLT-PLA-ESO-00000-0004, 2.0 (in preparation) — VLT Software Configuration Control Plan

[6] D.Lewine, 1991, O'Reilly & Associates, Inc. — POSIX Programmer's Guide

[7] B.Kernighan, D.Ritchie, 1988, Prentice Hall — The C Programming Language - Second edition

1.5 Glossary

The glossary used in this document is consistent with the IEEE definitions and with those in section 2.3.1 of SMP [3]. When a term is essential to understanding, the meaning is introduced in the section before it is used.

In addition, the following terms, related to Configuration Control matter, are used:

version
: A couple of numbers separated by a dot (`release.level`) identifying one of the implementation of the functionality of a software module, namely:

  `release` the progressive numbering of major changes in the module life. Although backwards compatibility is a general requirement, a new revision may represent changes in the module interface.

  `level` the progressive numbering inside a release of minor changes. A new level normally means changes in the implementation without involving the interface. Backwards compatibility with implementation having the same release number should be a design constraint.

  The version number applies to all files belonging to that implementation of the module. According to SCCS rules, the first implementation of a module has version 1.1.

<configuration control prologue>
: a commentary section at the beginning of each file containing all information (data, version, author, change explanation, etc.).

  The format will be defined in the issue of the present document.

SCCS
: a tool, normally present in any UNIX implemention, that allows control of write access to source files, and monitoring of changes made to those files. SCCS allows only one user at a time to update a file, and records all changes in a history file.

1.6 Abbreviations and Acronyms

The following abbreviations and acronyms are used in this document:

BNF	Backus Naur Form
CCS	Central Control Software
CRF	Change Request Form
DAT	Digital Audio Tape (DAT)
DFD	Data Flow Diagram
GUI	Graphical User Interface
HOS	High-level Operation Software
HW	hardware
IEEE	Institute of Electrical and Electronics Engineers (USA)
INS	Instrumentation Software

LCU	Local Control Unit
OLDB	On-Line Data Base
OS	Operating System
POSIX	Portable Operating System Interface
RDB	Relational Data Base
ROS	Remote Operations Software
SCCS	Source Code Control System (a UNIX tool)
SCCM	Software Configuration Control Manager
SQL	Structured Query Language
SW	software
TBD	To Be Defined
TCS	Telescope Control Software
VLT	Very Large Telescope
WS	Work Station

1.7 Editorial Conventions

The following styles are used:

bold
: in the text, for commands, filenames, pre/suffixes as they have to be typed.

italic
: in the text, for parts that have to be substituted with the real content before typing.

`teletype`
: for examples.

`<name>`
: in the examples, for parts that have to be substituted with the real content before typing.

bold and *italic* are also used to highlight keywords.

This page was intentionally left blank

2 Software Module

2.1 Overview

Strictly speaking, the definition of module does not belong to Programming Standards, but it is a useful concept for both technical and managerial reasons and many of the rules in the following sections use it.

This section defines a software module, the organization of the development and the documentation of the interface.

2.2 Definition Of A Module

A software module is a piece of software (code and documentation) able to perform functions and having an interface available to an external user to access the functions provided.

technically a module is a way to organize functions in homogeneous groups. The interface hides the implementation and system dependencies from the user.

managerially the module is the basic unit for planning, project control, and configuration control.

There is no rule to define how big a module shall be. Common sense and programming experience should be enough to identify what can be gathered and treated as a unique item. Examples of modules are: a driver for a specific board (the driver itself, install utility, configuration data file, etc.), the message system of CCS (library, utilities for debug and display, etc.), the sequencer (sequence interpreter, editor, etc.).

2.3 Module Name

Each module is identified by a name. The *module_name* can be made up to minimum two maximum six, suggested four, characters (a-z, 0-9) and shall be unique in the VLT project. Names equal or too similar to UNIX names shall be avoided. The case cannot be used to build different names: i.e., the following are referring to the same module: xyz, XYZ, xYz.

The *module_name* is used in the naming of all elements that belong to the software module according to the rules in section 3. The usage of the *module_name* in uppercase/lowercase depends on the type of element: file, procedure, type, etc.

The module name is defined during the design phase. In the Design Review the proposed names are checked against the existing names and, if ok, accepted. The module naming management is supported by a file, accessible by any user, listing the existing names and the attributes of each one (description, available version(s), status, error code range). The file is kept updated by the SCCM.

2.4 The Module Interface Description

A module offers services in the form of programs and/or procedures:

program can be the product of a compilation, such as a C-program, or the input to an interpreter (script file, SQL file, etc.). A program is invoked by means of a "run string" composing the name of the program and, optionally, parameters. The run string can be issued interactively by a user or programmatically by another process already running, e.g., from the start-up procedure. Each program shall be documented according to the format in section 2.4.3.

procedures (or routines), normally organized in librar(y/ies), are pieces of code, performing an independent action and available, via binding, to other entities. Each procedure shall be documented according to the format in section 2.4.4.

The services that a module offers can be accessed:

- by activating one of the programs provided by the module and passing information to it via one or more of the following: parameters in the run string, interactive input, setting of operating system environmental variables, writing/reading data in the OLDB, sending/receiving messages.
- by binding the routines provided by the module library to other programs.

2.4.1 General Formatting Rules

For each item, program or procedure, an independent entry in the documentation shall be provided. The documentation of each item is a list of sections, having a title and a text.

Similarly to UNIX documentation, NAME, SYNOPSIS and DESCRIPTION sections are always present and mandatory. Depending on the item type, other sections are present, some mandatory and some optional. Furthermore, on a module basis, other sections can be added according to specific need. Should this happen, it shall be stated in the Software Design Documentation.

If any mandatory section is empty, its header is kept with one of the following: "Not Applicable" or "None".

If any optional section is empty, its header is omitted.

A common remark. The one-line description, under the NAME section, should containing the essential keywords to identify the functionality performed because the **apropos** search feature of **xman** is based on such keywords.

2.4.2 Production of the Documentation

All the necessary information shall be present in the source files. Appropriate programs will be available to extract comments from source files (standard templates are defined later on in this document for both C-source and script file) or to format message or error definition files (standard templates will be defined in the CCS user manual). In case such programs are not available, the documentation shall be manually prepared according to the rules described here.

The extracted information is then formatted to be includable in the user documentation and to be displayable via "man/xman" utilities (**nroff** format). In both cases, the Unix Manual-like appearance is used.

Similarly to UNIX documentation, VLT man/xman pages will be organized in "sections". The rules for that will be provided in the next issue of the present document.

2.4.3 Program Template

program_name

NAME

program_name – brief description

SYNOPSIS

(How to invoke the program and the possible options.)

```
program_name  <opt1>  <opt2> . . . . .
```

DESCRIPTION

Detailed explanation of functionality and option(s).

FILES

The file(s) used by the program.
For each file: name, access (read, write, modify), description of the usage.

ENVIRONMENT

The environmental variable(s) accessed by the program.
For each variable: name, access (read, write, modify), description of the usage.

The OLDB table(s)/variable(s) used to exchange information with other modules (i.e., OLDB table(s)/variable(s) belonging only to the module shall not be documented here).
For each variable or group of related variables: name, access (read, write, modify), description of the usage.

COMMANDS

(The list of commands that the program can receive from other processes)
<COMMAND NAME> brief description of the action performed
<. . . . . >

RETURN VALUES

the possible return status and error values.

CAUTIONS

(optional) warnings, if any (i.e., special setup of hardware or software required, etc.)

EXAMPLES

(optional) examples, suggestions of how to use the program.

SEE ALSO

(optional) references to related information (other utilities, documents, etc.)

BUGS

(optional) known problems, if any, or planned improvements.

2.4.4 Procedure Template

procedure_name [2]

NAME

procedure_name – brief description of procedure

SYNOPSIS

The C binding (in the form of the ANSI C new style function definition).

If required, the header file(s) that must be included.

DESCRIPTION

Detailed explanation of functionality and arguments.

For each procedure argument a brief description, including allowed values and access (read, write, modify). Arguments shall be listed in the same order as in the synopsis.

FILES

The file(s) used by the procedure. For each file: name, access (open, close, read, write, update), description of the usage.

ENVIRONMENT

The environmental variable(s) accessed by the procedure. For each environmental variable: name, access (read, write, modify), description of the usage.

The OLDB table(s)/variable(s) accessed by the procedure and used to exchange information with other modules (i.e., OLDB table(s)/variable(s) belonging only to the module shall not be documented here). For each table/variable: name, access (read, write, modify), description of the usage.

RETURN VALUES

The possible return status and error values.

CAUTION

(optional) warnings, if any (i.e., special setup of hardware or software required, etc.)

EXAMPLES

(optional) examples, suggestions of how to use the procedure.

SEE ALSO

(optional) references to related information (other utilities, documents, etc.)

BUGS

(optional) known problems, if any, or planned improvements.

[2] (according to the naming convention, see section 3.)

2.5 Interface File

The interface file, a C-include file named *mod*.**h**, contains all the definition needed for the programmatic use of the module.

The include file is used by:

- the module user programmer(s) that find in it the definition of all procedures, type declarations, constants, etc., to perform consistency checks at compile time. Including the interface file it is not needed, neither allowed, to redefine locally, function prototypes, data type, etc.
- the module implementer programmer(s) to perform consistency checks between the external definition and the current implementation.

In detail, the interface file shall contain:

- header files used by the module implementation.
- constant declarations.
- data type declarations.
- functions prototypes, according to the so called ANSI "new style" format.
- possible message definitions, via inclusion of the definition file *mod***Messages.h** written according to CCS Message System syntax.
- possible error definitions, via inclusion of the definition file *mod***Errors.h** written according to CCS Error System syntax.

Template for interface file

file name: *mod*.**h**

```
<Configuration Control prologue>
#ifndef MOD_H
#define MOD_H

/*
 * Module MOD - Interface File
 */

/*
 * header files
 */
/*none*/

/*
 * constants
 */
#define modMAX_TIME_TO_WAIT   100

/*
 * data types
 */
typedef enum {
             modSTART_MOTOR=1,
             modSTOP_MOTOR,
             modREACH_PARK_POSITION
} modWHAT;

typedef struct {
           ........;
           ........;
} modTIME;


/*
 * macros
 */
# include modMacros.h


/*
 * functions prototypes
 */
integer modDoAction (modWHAT *modAction, modTIME *modWhen);
integer modWait (int modTimeToWait);

/*
 * message definitions
 */
```

```
#include modMessages.h

/*
 * error definitions
 */
#include modErrors.h

#endif /*!MOD_H*/
```

2.6 Directory Structure

The complete directory structure of the VLT software is not within the scope of this document, but is part of the System Software Design Description document (see [3] 4.2.6).

Appendix C provides a preliminary description of the VLT Directory Structure.

2.6.1 Module Directory Tree

For development and management reasons there is a need to have the possibility to access several different versions of the same module. Each version has the same structure but a different root (*mod*`ROOT` stands hereunder as the full path of a root).

REMARK: including the appropriate *mod*`ROOT` in your **path** definition, allows you to switch easily from one environment to another.

Each area shall have the the following minimum subdirectory tree:

```
<modROOT>/
          |------include/  (*)
          |------src/      (*)
          |------object/          (optional)
          |------oldb/     (*)    (optional)
          |------forms/    (*)    (optional)
          |------doc/
          |------test/     (*)
          |------bin/
          |------lib/
          |------man/
          |------tmp/
          |------.../             (optional)
          |------.../             (optional)
```

(*) these directories have the `SCCS` subdirectory.

The content of each subdirectory is:

`include/` (**)
: include files

`src/`
: C-source files and scripts.

`object/` (**)
: object files, only for VxWorks applications.

`oldb/` (**)
: OLDB definitions, if any.

`doc/`
: documentation files, at least the User Manual and the Maintenance Manual (source and printable formats).

`test/`
: test software: source, executable, input data, reference output data, etc.

`bin/` (**)
: scripts and executables (used during development and test).

`lib/` (**)
: libraries (used during development and test).

`man/` (**)
: on-line documentation

`tmp/`
: local scratch area for temporary files (e.g., during test).

`.../`
: any other module specific directory.

(**) the contents of these directories are copied at installation time to the system directory tree.

2.7 README

For each module a README file shall be present in `src/`.

The README file shall contain a brief description of the module.

3 Naming Conventions

Naming conventions help:

- make identification of objects easier
- improve consistency across the system
- avoid conflicts

This section defines naming rules for:

- Files:
  - Path
  - Filename
  - Filetype
- C-language items:
  - Functions
  - Macros
  - Constants
  - Variables
  - Data type
  - Structure and union members
  - Enumerators
- Operating system environmental variable

Rules to name the following will be provided in the CCS documentation.

- On-line Data Base table
- On-line Data Base variable
- Commands
- Error description

Naming conventions are not applicable to special objects, like Makefiles, README, etc., for which naming rules are already established.

3.1 General Rules

Each name must not exceed 31 characters long and, considering uppercase and lowercase letters being equivalent, must be unique in the VLT scope. The allowed set of characters is: a-z, A-Z, 0-9, "_" (low line or underscore).

Each name is formed by two parts:

```
[<master_area>]<description>
```

master_area

intended to organize names in a one-level hierarchy system. The grouping criteria are in the following table:

for objects that belong to	*master_area is*
the same software module	the module name (see 2.3)
the same physical part of the VLT system (can be an equipment or a smaller part)	the part name (refer to the HOS/Access Configuration and Control documentation for the rules for naming physical parts)
the whole VLT	"VLT" (but can be omitted, if there is no ambiguity with other environments)

description

describes the content of the object with which the name is associated. As a general requirement, it must be meaningful and readable. Abbreviations should be avoided

The examples hereunder use a module called MOD and an equipment called T1.

3.2 File

A file is identified by:

```
<path><file_name>.<file_type>
```

3.2.1 Path

The path shall always be **relative** to an environmental variable that represents the logical root of the domain to which the file belongs.

3.2.2 File_name

The *description* should represent the content/role of the file.

No restriction in the use of lowercase/uppercase and underscore, though uppercase words should only be used for acronyms or aliases.

A file can belong to: a module, an equipment, the whole system.

For some files additional conventions are defined:

1. files containing C source code the following rules shall be applied:

   if a C-source file contains the `main()`, the file name is the program name. This rule, when applicable, supersedes the following rules.

   if a C-source file contains a single procedure, the file names matches the procedure name.

   if a C-source file contains more than one procedure [3], the file name comprises the significant words common to all procedure names in the module.

[3] This case should be used only when procedures are closely related and are very simple. The general rule is one procedure per file (see 4.2.2).

2. header files of a module: *mod*.h, *mod*Errors.h, etc.

Examples:

```
mod.h
modHelp
mod.c
modTerminalManagement.c
modControlExposure.c
modTestInputData
ChangeNotice
T1_Error.log
VLT_standard_copyright
```

3.2.3 File_type

The file type identifies the type of file, e.g., c-source, include, LaTeX file, VLT defined standard type, etc.

The following file types are mandatory and their use is restricted to the specified cases:

empty	executable
`.c`	C language source
`.h`	C language include (or header)
`.o`	object code
`.a`	library
`.tex`	LaTeX source
`.ps`	Postscript file

The use of specific software tools can impose additional rules.

3.3 C-language Items

REMARK: VxWorks, OSF/MOTIF and RTAP are going to be used for development. Although similarities exist among the three, each one uses a different naming style. The following standard does not completely follow any of them.

When used, the "master_area" is the "module_name" and is always lowercase.

3.3.1 Functions

description is formed by two parts: the action (a verb) and the object (a noun) of the action. Each word forming the *description* is capitalized.

```
modStartExposure
modMoveSlit
modCloseSession
modReadTachoSpeed
modSetTime
```

A function can belong to only one module. Theoretically, there are two types of functions, depending on the external accessibility to them:

public: functions accessible by other modules, i.e., functions belonging to the programmatic interface of the module, and documented in the User Manual.

private: functions accessible only by other module functions, i.e., functions not belonging to the programmatic interface of the module and documented in the Maintenance Manual.

3.3.2 Macros

Macros with parameter(s) follow the same convention as for a function (see 3.3.1).

Macros without parameters follow the same convention as for the constants (see 3.3.3).

3.3.3 Constants

description is the literal representation (THOUSAND) or the logical meaning (MAXCOLUMN) of the constant.

The *description* is in UPPERCASE, multiple words are separated by an underscore (_).

A constant can belong to a module or be of general use.

Examples:

```
modMAX_EXPOSURE_TIME
modSLIT_X_PARK_POSITION
modSLIT_Y_PARK_POSITION

TIME_TO_START
```

The POSIX constants (such as NULL, TRUE, FALSE, etc.) and the ones defined in the general VLT definition file ("vltstd.h") shall NOT be redefined at application level.

3.3.4 Local Variables

master_area is empty.
The *description* is the content of the variable. Each word except the first one is capitalized:

Examples:

```
exposureTime
slitPreviousPosition
counter
sessionId
tachoSpeed
time
```

3.3.5 Global Variables

The *description* is the content of the variable. Each word is capitalized:

Examples:

```
modExposureTime
```

```
modLocalTime
```

3.3.6 Data Type

The *description* is the type identification. Types can be elementary types (integer, char, etc) or complex, like structures and unions.

The *description* is in UPPERCASE, multiple words are separated by an underscore (_).

A type can belong to a module or be of general use.

Example:

```
modSLIT_POSITION
```

The POSIX types (such as FILE, etc.) and the ones defined in the general VLT definition file ("vltstd.h") shall NOT be redefined at application level.

3.3.7 Structure Members and Union Members

Same as for local variables (see 3.3.4).

3.3.8 Enumerators

Same as for constants (see 3.3.3).

3.4 Operating System Environmental Variable

The *description* shall represent the content of the variable.

The whole name is in UPPERCASE. Different words are separated by an underscore (_)

An operating system environmental variable can belong to: a module, an equipment, the whole system and be used across different applications (do not confuse with local variables used inside script files, see 5.3.3).

Examples:

```
MOD_ROOT
T1_STARTUP_MODE
VLT_SYSTEM_ROOT
```

Some variables (such as HOME, USER, etc.) are already defined by the operating system and others by the general VLT startup procedure and can not be redefined at application level.

This page was intentionally left blank

4 C Language

This section is applicable to all C-language code, both for UNIX and VxWorks target environments. Should differences exist, the two cases are treated separately.

4.1 C Compiler Version

4.1.1 UNIX Applications

All VLT code written in C shall conform to the ANSI C standard ([4]) 7.1.3) as defined in:

[1] Programming Language C, ANSI Standard X3.159-1989.

More information can be found in:

B.Kernighan, D.Ritchie, 1988, Prentice Hall — The C Programming Language - Second edition [7].

TBD: qualified compilers for each development platform (SUN and HP) and the standard run string with the default options.
At present the GNU C Compiler version 1.39 shall be used

4.1.2 VxWorks Applications

GNU C Compiler as from the standard VxWorks version.

TBD: standard run string with the default options.

4.2 File Templates

For C development, files can be grouped into two categories: source files (.c) and include files, also called header files (.h).

The source file can contain:

- a single procedure (a special case is the *main()*).

- a group of closely related procedures, each of which would be too small for an independent file.

For those functions that form a library, it is recommended to have one file per function. The benefit is that the executables that link with such a library will take from the library only the modules that are really necessary.

As a rule of thumb, each file should range between three to ten pages.

4.2.1 Source file (.c) - main

Each *main* requires a separate file called *program_name*.**c**. Only signal catching functions and local functions can be inserted in this file. Any function, structure, etc. belonging to a software module shall be accessed including the appropriate include file.

Each main source file shall be structured according to the template at page 25.

4.2.2 Source file (.c) - function/s

A function, also called a procedure or subroutine, is an elementary piece of code provided to other user programs. The interface of each function is defined by the function name, the parameter(s), the return value and by the effect(s) that the execution of the function has on the computing environment.

Each source file containing one or more function(s) shall be structured according to the template at page 28.

Template for main

file name: *program_name*.**c**

```
<configuration control prologue>
/*****************************************************************************
*   NAME
*       <program_name> - <brief description of program>
*
*   SYNOPSIS
*       <program_name> [<par_1> [<par_2>  . . . [<par_n>]]]
*
*   DESCRIPTION
*       <detailed explanation of the functionality>
*        ..........................................
*        ..........................................
*        ..........................................
*                .
*                .
*        ..........................................
*
*   <par_1> <description and grammar rules for the first parameter>
*            ................................
*   <par_2> ................................
*                .
*                .
*   <par_n> <description and grammar rules for the n-th parameter>
*            ................................
*
*   FILES
*       <file_1>  <access> <meaning and purpose of the file>
*                          ..............................
*       <file_2>  <access> ..............................
*        ......
*
*   ENVIRONMENT
*       <var_1>   <access> <meaning and purpose of the variable>
*                          ..............................
*       <var_2>   <access> ..............................
*        ......
*
*   COMMANDS
*       <Command_1>  brief description of the action performed
*                     ....................................
*       <Command_2>  ....................................
*        ......
*
*   RETURN VALUES
*       <ret_value_1> <message> <diagnostic of the error>
*                               ......................
*       <ret_value_2> <message> ......................
*        ......
```

```
*
*   CAUTIONS (optional)
*       ......
*       ......
*
*   EXAMPLES (optional)
*       ......
*       ......
*
*   SEE ALSO (optional)
*       ......
*       ......
*
*   BUGS     (optional)
*       ......
*       ......
*
*------------------------------------------------------------------------
*/
#define POSIX_SOURCE 1

/*
 * System Headers
 */
#include <.....>
#include <.....>
.......

/*
 * Local Headers
 */
#include "........."            /* ........ */
#include "........."            /* ........ */

/*
 * Signal catching functions
 */
.......
.......
.......

/*
 *Local functions
 */
.......
.......
.......


/*
 * Main
```

```
 */

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

/*___oOo___*/
```

Template for single procedure or group of procedures

file name: *procedure*.**c** or *procedure_family*.**c**

```
<configuration control prologue>
/*****************************************************************************
*   NAME
*       <name>{[,<name>]} - <brief description of procedure(s)>
*
*   SYNOPSIS
*       <ret_type> <name> (arg1, arg2, ..., arg-n)
*
*       [<ret_type> <name> ( ... )]
*
*       #include "......h"
*
      (more than one function can be described here -
        see ad example the man page of documentation of "printf")
*
*
*   DESCRIPTION
*       <detailed description of the function(s) performed>
*        ...........
*        ...........
*        ...........
*        ...........
*
*   <arg1> <description and grammar rules for the first argument>
*           ..................................
*   <arg2> ..................................
*                .
*                .
*   <arg-n> <description and grammar rules for the n-th argument>
*           ..................................
*
*   FILES
*       <file_1>  <access> <meaning and purpose of the file>
*                          ..............................
*       <file_2>  <access> ..............................
*        ......
*
*   ENVIRONMENT
*       <var_1>   <access> <meaning and purpose of the variable>
*                          ..................................
*       <var_2>   <access> ..............................
*        ......
*
*   RETURN VALUES
*       <ret_value_1> <message> <diagnostic of the error>
*                               ......................
*       <ret_value_2> <message> ......................
*        ............
```

```
*
*   CAUTIONS (optional)
*       ......
*       ......
*
*   EXAMPLES (optional)
*       ......
*       ......
*
*   SEE ALSO (optional)
*       ......
*       ......
*
*   BUGS     (optional)
*       ......
*       ......
*
*------------------------------------------------------------------------
*/
#define POSIX_SOURCE 1

/*
 * System Headers
 */
#include <.....>
#include <.....>
.......

/*
 * Local Headers
 */
#include "........."            /* ........ */
#include "........."            /* ........ */
.......


/*
 * Function definition
 */
<ret_type> <name>(
                  <typ1> <arg1>,
                  <typ2> <arg2>,
                  ....)
{
..... code ....
..... code ....
..... code ....
..... code ....
..... code ....
}
```

```
...... if any, 2nd function and so on .....


/*___oOo___*/
```

4.2.3 Include file (.h)

The following rules are mandatory:

1. Include files must not contain executable code.
2. In order to ensure that it will be included only once, wrap each include file in the following way:

   file: *name.h*

   ```
   #ifndef NAME_H
   #define NAME_H
   .....
   .....
   #endif /*!NAME_H*/
   ```

3. Provide an in-line comment for each item in the header file.
4. Organize the header file in sections of homogeneous items: constants, macros, function prototypes, type declarations, etc.
5. ANSI-C excludes the use of a **sizeof()**, a cast, or an enumeration constant in *constant-expressions* of conditional compilation(**#if**).
   About the use of **sizeof** see also 4.7 3.
6. Use only **#ifdef** (or **#if define**) to check whether a symbol is defined.

When using header files:

– use explicit < > brackets for system include file (POSIX) and quoted form for local file.

```
#include <stdio.h>
#include "mod.h"
```

– do not use full pathname for **#include** directives, but the compiler directive **-I**.
NO ABSOLUTE PATH IS ALLOWED.

The complete header file template follows:

```
#ifndef NAME_H
#define NAME_H
<configuration control prologue>
/*****************************************************************************
*   <name.h> - <brief description of the purpose of this .h file>
*               ................................................
*               ................................................
*
*----------------------------------------------------------------------------
*/

#define _POSIX_SOURCE 1

/*
 * ........
 */
.....................           /* ........ */
.....................           /* ........ */
.....................           /* ........ */

/*
 * ........
 */
.....................           /* ........ */
.....................           /* ........ */
.....................           /* ........ */


#endif /*!NAME_H*/
```

4.3 Source Readability

The following are guidelines.

4.3.1 Indentation and Spacing

1. write only one statement per line.

   Exceptions can be accepted when the repetition of the same group of statements per line improves readibility:

   ```
   argv[i++] = 2; argv[i++] = 99;  /* set the even and odd elements */
   argv[i++] = 4; argv[i++] = 97;
   ```

2. use blanks around all binary operators, except "." and ".->";

3. in *compound statements*, put braces ({}) in a separate line and alligned with indented statements;

   ```
   {
   ......;
   ......;
   }
   ```

   If a compound statement is more than 20 lines, the closing brace should have an in-line comment to indicate which block it delimits.

   ```
       .
       .
   }     /* ...... */
   ```

4. use a blank after commas (arguments list, values, etc.), colon, semicolon and control flow keywords:

   ```
   procedure(arg1, arg2);
   for (i = 0, j = strlen(s) - 1: i<j; i++, j--)
   if (....)
   while (....)
   ```

5. do not use blanks between an *identifier* and any of "(", ")", "[", "]":

   ```
   procedure(arg1, arg2);
   a[i] = 1;
   z[i] = a + (sin(x) + cos(b[i]))
   printf("%d", (a + b))
   ```

6. line up continuation lines with the part of the preceding line they continue:

   ```
   a = (b + c) *
       (c + d)

   if ((a == b) &&
   ```

```
    (c == d))
    ...

printf("%d %s",
       (a + b),
       "abcd")
```

7. The indent step is 4 characters. The following templates [4] show how to indent the most frequent structures:

   (a) **typedef struct**

```
typedef struct {
    ...... ......;
    ...... ......;
    ...... ......;
} ......;
```

   (b) **if**

```
if (....)
    {
    ......;
    ......;
    }
else
    {
    ......;
    ......;
    }
```

   (c) **else-if**

```
if (....)
    {
    ......;
    ......;
    }
else if (....)
    {
    ......;
    ......;
    }
else if (....)
    {
    ......;
    ......;
    }
     .
     .
else
    {
    ......;
```

[4] The proposed style corresponds to the one used by many source code analyzers.

```
        ......;
        }
```

(d) **switch**

```
    switch (....)
        {
        case ...:
            ......;
            ......;
            break;

        case ...:
            ......;
            ......;
            break;
          .
          .
        default:
            ......;
            ......;
            break;
        }
```

or, if suitable:

```
    switch (....)
        {
        case ...: ......; ......; break;
        case ...: ......; ......; break;
          .
          .
        default: ......; ......; break;
        }
```

(e) **while**

```
    while (....)
        {
        ......;
        ......;
        }
```

(f) **for**

```
    for (...; ...; ...)
        {
        ......;
        ......;
        }
```

(g) **do-while**

```
    do
        {
        ......;
        ......;
```

```
    }
  while (....);
```

4.3.2 Comments

Both block-style comments and in-line comments are acceptable.

1. block-style comment

```
/*
 * ..................................
 * ....................................
 * ............................
 */
```

   Each block is preceded by an empty line and has the same indentation as the section of code to which it refers.

   A block-style comment should always appear at the beginning of each relevant segment of code.

2. in-line comment

```
.....................          /* ........ */
.....................          /* ........ */
.....................          /* ........ */
```

   Brief comments on the same line of the statement that they describe. They can start on columns (33+n*4) n=0, 1, 2, 3... (33, 37, 41, 45, etc.). There must be at least 4 spaces between the code and the start of the comment. Comments on contiguous lines must be aligned on the same column.

   In-line comment should not be broken in multiple lines:

```
.....................          /* ........          AVOID THIS!
                                   ........... */
```

   Use in-line comment to document variable usage and other small comments. Prefer block-style comments to describe the computation process.

4.3.3 Naming Conventions

See section 3.3.

4.4 Allowable Data Types

TBD. A set of authorized types will be defined after the implementation of CCS.

4.5 Language Sensitive Editor

TBD

4.6 Forbidden Instructions

The "old style" function declaration:

```
power (base, n)
int base, n;
    {
    .......
    }
```

is not allowed anymore. Use instead ANSI-C style (see [7] page 26 and page 217 A8.6.3):

```
int power (int base, int n)
    {
    .......
    }
```

4.7 Mandatory Practices

1. Structures and unions must always be passed to or returned by functions using pointers.
2. global variables are not allowed outside the scope of the module.
   In case of shared libraries the restrictions with global "initialized data" shall be taken into consideration. When building a shared library it is recommended to segregate the declarations of exported initialized data from the sources for each object, and to place them in separate source files.
3. Let the compiler work out lengths: use **sizeof** rather that declaring the length of a structure or of an array explicitly.
   About the use of **sizeof** in include file see also 4.2.3 5.
4. Use **cast** explicitly when conversion is necessary.
5. Access to data shall be independent of its physical size and alignment.
6. Provide a **default** for **switch** statement. Even when you expect one of the cases to catch all answers, have it diagnose as a catastrophic error if the condition was not expected.
7. Assumptions should not be made about the order of parameter evaluation. Always explicitly code your desired order of evaluation of subexpression. Some machines evaluate the parameters from right to left, others evaluate them from left to right. For example:

```
func(x * i, y / i, i++);
```

8. To avoid machine-dependent fill methods, do not apply the right-shift (>>) or left-shift (<<) operators to signed operands, and do not use shifts as a divide/multiply.

9. Uninitialized automatic variables shall never be used. 'lint' (see B) will detect them.

10. Data types must not be mixed across function boundaries. Type **cast** should be used.

11. Function prototypes, constant, type definitions, etc. are accessed including the appropriate **.h** file. Redefining them is not permitted.

12. If the return value of a function would never be used, the function shall be declared as a **void** function.

13. Pointer assignment shall be between pointers with the same data type. Avoid treating pointers as integers or vice-versa.

14. To check the validity of a pointer compare it to `cast(NULL)`. Do not use the integer 0.

4.8 Suggested Practices

1. Beware of side effects in macro parameters. Don't use auto increment or decrement on arguments to a macro; if the macro references the argument more than once, then the increment/decrement happens again also.

2. Avoid the use of **goto** statements. If necessary, clearly document it.

3. Limit the use of global variables inside a module. (It does not apply to VxWorks where by definition global variables of a module are common to the whole LCU).

4. Break each **case** of the **switch** statement to avoid falling through to the next case. If unavoidable, clearly document it. (The only exception is multiple **case** labels.)

5. Avoid using **char** variables to store non-character data.

6. Expect the unexpected: Provide an else for if statements. Even when you expect one of the cases to catch all answers, have it diagnose as a catastrophic error if the condition was not expected.

7. Use parentheses even when not required to improve clarity. Using parentheses also helps avoid errors caused by misunderstood operator precedent.

8. If the same index addresses two or more data then define an array of structures instead of separate vectors.

9. Don't use realloc() with a NULL pointer, a pointer that has not been allocated, or one that has been freed. You may get away with it on some systems, but on others it may cause problems that won't be detected until your customer has trusted the system with a lot of data.

10. Do not increment/decrement loop control variables in the middle of the block. Otherwise it makes it harder for you to find the loop control, and you may miss the increment, or increment past the boundary, under certain conditions.
You should never have an additional increment/decrement inside a for loop. In a while loop, put the increment/decrement at the top or bottom of the block

11. When you have a software error condition, print out the file and line number in the message. This is facilitated by __LINE__ and __FILE__, which are defined by the compiler to be the current line number and file name:

```
fprint(stderr, "Fatal: bad pointer, line %d, file: %s\n",
       __LINE__, __FILE__);
```

12. Provide an in-line comment for each declaration of *variable* (except for obvious loop counters like i, j, etc.), *type*, **#include** (except POSIX system files like **stdio.h**, etc.), etc.

13. Avoid debug statements. They are rarely useful (it always turns out you want to look at other or more variables than the ones you are printing), they are only used at the very beginning of the lifetime of a program, they make the source file longer and more difficult to read, and modern debuggers are by far more efficient and versatile.
    If unavoidable, use conditional compile to turn them on/off as shown:

```
#ifdef DEBUG
/* debug print statements */
#endif
```

14. Setting up of enumerated variables starts with 1, so uninitialized values are easily detected. Example: a failure to read the traffic light or possibly even to call the read_traffic_light function will be detected:

```
typedef enum {
    CT_RED=1,
    CT_YELLOW
    CT_GREEN
} COLOR_TYPE;

{
COLOR_TYPE color = CT_INVALID;

color = readTrafficLight();

switch (color)
    {
    case CT_RED:
        stop();
        break;

    case CT_YELLOW;
        stopIfPossible();
        break;

    case CT_GREEN;
        cruise();
        break;

    default:
        fprintf(stderr, "\nFatal: bad pointer, line %d,
                file: %s\n", __LINE__,__FILE__);
        abort();
    }
}
```

4.9 Error Return

See CCS/ErrorSystem documentation.

5 Script Files

5.1 Standard Shell

A standard shell command language is not yet defined. The Part 2 of the POSIX standard (IEEE Std. 1003) is planned to specify a standard shell based on the System V shell with some features from the C shell and the Korn shell. The standard includes also the User Portability Extension (UPE) that covers utilities like **vi**, **make**, **man**, **mailx**, etc. (see [6] page 3).

For the VLT Software the following have been chosen:

– **Bourne shell** as the mandatory shell for scripts.

– **Korn shell** as the suggested interactive shell

5.2 File Template

As a rule of thumb, each file should range between two to five pages.

Each script file shall be structured according to the follow template:

Template for script file

file name: *program_name*

```
#! /bin/sh
<configuration control prologue>
#************************************************************************
#   NAME
#       <program_name> - <brief description of program>
#
#   SYNOPSIS
#       <program_name>  [[[<par_1>] <par_2>] ...]
#
#   DESCRIPTION
#       <detailed explanation of the function performed>
#        ...........
#        ...........
#        ...........
#        ...........
#
#   <par_1> <description and grammar rules for the first parameter>
#             .................................
#   <par_2> .................................
#                    .
#                    .
#   <par_n> <description and grammar rules for the n-th parameter>
#             .................................
#
#   FILES
#       <file_1>  <access> <meaning and purpose of the file>
#                           ..............................
#       <file_2>  <access> ..............................
#        ......
#
#   ENVIRONMENT
#       <var_1>  <access> <meaning and purpose of the variable>
#                          ................................
#       <var_2>  <access>  ................................
#        ......
#
#   RETURN VALUES
#       <ret_value_1> <message> <diagnostic of the error>
#                                .......................
#       <ret_value_2> <message> .......................
#         ......
#
#   CAUTIONS (optional)
#       ......
#       ......
#
#   EXAMPLES (optional)
#       ......
```

```
#       ......
#
#   SEE ALSO (optional)
#       ......
#       ......
#
#   BUGS     (optional)
#       ......
#       ......
#
#------------------------------------------------------------------------
#
     <signal trap (if any)>
#
#
#

   ..... code ....
   ..... code ....
   ..... code ....
   ..... code ....
   ..... code ....

#
# ___oOo___
```

5.3 Readability

The following are guidelines.

5.3.1 Indentation and Spacing

Do not put more than one instruction on the same line.

The standard indentation step is 4. The style is the following:

```
<command-list>


if <command-list>
then
    <command-list>
    <....>
else
    <command-list>
    <....>
fi


for <name>     (or:  for <name> in <word> . . .)
do
    <....>
done


while <command-list>
do
    <....>
done


case <word> in
    ....): <....>
        <....>
        ;;

    ......): <....>
          <....>
        ;;

    *): <....>                  # REMARK: the default label
        <....>                  #         is mandatory.
        ;;

esac
```

5.3.2 Comments

Both block-style comments and in-line comments are acceptable.

1. block-style comment

   ```
   # ..................
   # ..................
   # ..................
   ```

   is one or more line having # as the first non-blank character. Each block is preceded by an empty line. Text is aligned with the code that it documents.

   A block-style comment should always appear at the beginning of each relevant segment of code.

2. in-line comment

   ```
   ..................      # ..........
   ..................      # ..........
   ..................      # ..........
   ```

   Brief comments on the same line of the statement that they describe. They can start on columns (33+n*4) n=1,2,3... (33, 37, 41, 45, etc.). There must be at least 4 spaces between the code and the start of the comment. Comments on contiguous lines must be aligned on the same column.

   Limit the use of in-line comment to document variable usage and other small comments. Prefer block-style comments to describe the computation process.

5.3.3 Local Variable Naming

Local variables can be uppercase or lowercase. The suggested use is:

– UPPERCASE objects (file names, etc.)

```
SOURCE_DIR=$CURRENT_MODULE/source
INPUT_FILE=$1
```

– lowercase: variables used in computation (index, counter, etc.)

```
read a
for file in ...
```

Being local, no restriction in naming is defined. Provided that each name is meaningful and understandable, avoid abbreviation.

In any case the use of the name of a environmental variable for a local scope variable is strongly discouraged, even when the global variable is not used in the script.

5.4 Forbidden Instruction/commands

No other control flow operator is permitted in addition to those listed in section 5.3.1.

Local redefinition of OS and VLT variables shall be clearly documented and cannot be exported to global environment.

5.5 Mandatory Practice

Always provide the default ***):** pattern in the **case** statement.

5.6 Suggested Practice

Multiple **case** labels should be placed on separate lines.

5.7 Returns Values

Use only one exit point for all possible successful end(s). Write explicitly **exit** or **exit 0** when there are other possible exit values for abnormal termination.

Use **exit n** to exit in abnormal situation. Document each case in the script header (RETURN VALUES section).

6 Make

6.1 Make Version

To avoid discrepancies with the different vendor implementations of *make*, the **GNU make 3.59** will be used.

6.2 Makefile Structure

A **Makefile** file shall exist in every directory containing sources, in particular:

mod_name/: with the entries:

all to regenerate software, documentation and test.

install to install software, documentation and test

clean to clean up the development areas

`source/`: with the entries:

all to regenerate all the software

install to move executable, libraries, include and script files to target directories

clean to clean up the software development areas

`doc/`: with the entries:

all to regenerate all the documents (from source, .tex file, etc.) both as printable form and in the man/xman format

install to move man/xman files to target directory

clean to clean up the document development areas

`test/`: TBD

Standard templates and the use of **make** in conjunction with **SCCS** will be provided in the next issue, after the definition of the CC structure and of the target system file organization (see also appendix C).

This page was intentionally left blank

7 UNIX

The development operating system standard for the VLT shall be **UNIX System V** ([4] 7.1.2 1a).

7.1 Callable Interface for System Calls

According to section 7 of Software Requirement Specification [4], only POSIX calls are accepted. The reference document for POSIX calls is [2]

IEEE Std. 1003.1-1990, Information Technology—Portable Operating Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]

As a program development help, [6] POSIX Programmer's Guide, by D.Lewine, 1991, O'Reilly & Associates, Inc. can be used. This book presents the recommendations of the IEEE standard to operating system developers, in a form better suited to application developers.

The use of non-POSIX calls must be proved necessary during the design phase and is subject to ESO approval. In such a case, all calls to OS implementation dependent routines shall be put in a separate file, providing interface routines to access them. The other code uses the system dependent part ONLY via the interface routines.

For interprocess communication and data sharing, applications shall use the Central Common Software. The use of other mechanisms must be proved necessary during the design phase and is subject to ESO approval. In such a case, all the specific functions shall be implemented as a separate software module, providing a programmatic interface to access it. The other code can use the alternative mechanism ONLY via the interface of such a module.

7.2 OS Utility

System V.

7.3 Signal Treatment

TBD

This page was intentionally left blank

8 VxWorks

No limit to the use of the functions currently part of the VxWorks. When available, POSIX.1 and real time extention POSIX.4 calls should be used whenever possible.

When not POSIX compliant, calls to OS implementation dependent routines should be put in a separate file, providing interface routines to access them. The other code uses the system dependent part ONLY via the interface routines.

The LCU Common Software services, like interprocess communication and data sharing, shall be used. The use of other mechanisms must be proved necessary during the design phase and is subject to ESO approval.

This page was intentionally left blank

APPENDICES

This page was intentionally left blank

A Media for Deliverable Software

Digital Audio Tape (DAT) cartridge shall be used to deliver any type of software and documentation internally between ESO and contractors, in both directions.

When available and with the approval of ESO, network link(s) can be used in addition or as an alternative to DAT.

A.1 Media Format

Only DAT/DDS cartridge (90 m.) UNCOMPRESSED format are allowed.

A.2 Media Labeling

Each media item shall have a label, with a layout as in figure 1, having the following information:

<**date**> date of writing

<**content**> brief explanation of the content (e.g., name of the software module, documentation for Design review, etc.).

<**reason**> reason to produce the tape (acceptance, bug fix, information, update, etc.)

<**from**> identification of the sender (name, Company)

<**system**> operating system (name and version) and computer (model) used in the generation

<**command**> command issued to create the tape

<**pwd**> the current directory at the time the command was issued

REMARK: <**date**> and <**content**> shall be written on the spine label too.

A.3 Software Format

All delivery shall be one *tar-file* (or *compressed tar-file*) produced using the **tar** command (and **compress**).

tar-path names shall be relative (i.e., path names shall not begin with a '/').

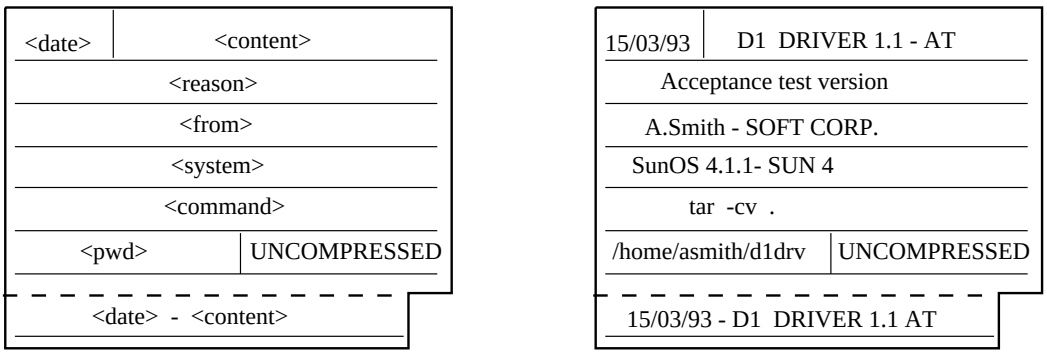

Figure 1: Tape Label Layout

B Test Tools

At present, several tools are under evaluation to cope with:

- static checks (lint)

- consistency checks (application of this standard, cross-check among files)

- software quality measure

- test coverage

- standard test environment

TBD in the next version of this document or in separate document, the selected tools and the rules to use them.

This page was intentionally left blank

C VLT Directory Structure

The complete directory structure of the VLT software is not within the scope of this document, but is part of the System Software Design Description document (see [3] 4.2.6).

This section provides a preliminary description of the VLT Directory Structure. Its main purpose is to provide a logical structure for the use of **make**.

The following logical areas are defined:

the development area of a module: this area shall have the structure described in 2.6.1 and is the responsibility of the programmer. No reference to such areas can be made from other modules.

Development areas can be created on any machine and in multiple copies. The node-directory of the root of each area is the identification for each area.

Any activity in the development area of a module shall be based on the standard structure and all paths shall be relative to the module root. Reference to software belonging to other modules shall be done via the environmental variable **VLTROOT** (see later), appropriately defined to the needed operating environment.

the software archive: there is only one archive area under the responsibility of the Software Configuration Control Manager. The archive contains the whole history of the VLT software and is used as follows;

1. the software is developed and tested under a development area.
2. after being accepted, the software is checked-in to the archive by the responsible of the development.
3. All source files are transferred from the development area into the corresponding subdirectory in the archive (**modx/...**).
4. all source file are checked-in to SCCS.
5. a new area is created according to the version number (**modx/n.m**)
6. all files are checked-out for read from SCCS and executables are regenerated from scratch in **modx/...**.
7. **include/**, **object/**, **oldb/**, **bin/**, **lib/**, **man/** are copied under **modx/n.m** copied by the SCCM.

The archive is used to install the software in the target machines.

```
VLTARCHIVE/
          |-----modx/
          |          |------include/
          |          |              |---.SCCS/
          |          |
          |          | . . . (complete structure)
          |          |
          |          |---1.1/
          |          |      |------include/
          |          |      |------object/
          |          |      |------oldb/
          |          |      |------bin/
          |          |      |------lib/
          |          |      |------man/
          |          |
          |          |---1.2/
          |          |      |------include/
          |          |      .
          |          |      .
          |          |      |------man/
          |          .
          |          |---n.m/
          |                 .
          |
          |-----mody/
          |          |------include/
          |          |              |---.SCCS/
          |          |
          |          | . . . (complete structure)
          |          |
          |          |---1.1/
          |          |
          |          |---1.2/
          |          .
          |          |---n.m/
          .
          .
          .
          |
          |----forms/
                     | (To Be Defined)
```

the operating area: contains the software that can be used. It is the target for **make install** and is in the search path for libraries and include files when generating a module.

Operating areas are defined both on target computers and on development computers. For target computers the installation is from the archive. For development computers, the area is first created from the archive and then updated from the development area for integration activities involving more modules.

It is possible to create many operating areas and, appropriately defining the environmental variable **VLTROOT**, to switch from one to another.

```
VLTROOT/
       |----etc/
       |       |----include/
       |       |------object/
       |       |------oldb/
       |       |------bin/
       |       |------lib/
       |       |------man/
       |
       |----xyz/
       |       |----include/
       |       .
       |
       |--forms/
       |       .
       |
       |--<data>
       .
       .
```

___oOo___