northern
telecom

Approved: *Evidence (signatures or cocos) has been submitted that Issue 1.5 of this document has been ap**proved.***

C Coding Standard

Standards Specification

Author(s):	Allen Aubuchon, Adrian Brandt, Michael Giertych, Eric Hildum, Mikhail Khodosh, Mary Lang, Khai Nguyen, Siamak Ashrafi
Manager(s):	Rod Bagg Mary Lang
Dept:	4Q22, 4R32, 4K31, 4K43

Date:	April 14, 1995
Issue:	1.5
Project Name:	SL-1
Keywords:	standard, C, code, coding, portability

Abstract: This document describes the C coding standard for SL-1 projects.

Approvals

C Coding Standards

Issue 1.5

Rod Bagg, Manager, 4K42, Mt. View, USA Date

Ramiz Ballou, Release Management Manager, 4K16, Mt. View, USA Date

Approved: *Evidence (signatures or cocos) has been submitted that Issue 1.5 of this document has been ap**proved.***

Revision History

ISSUE NO.	DATE	AUTHOR(S)	REASON FOR ISSUE
0.1	11/01/90	M. Khodosh	First Draft
		M. Giertych	
		A. Aubuchon	
1.0	11/08/90	M. Khodosh	First Release
1.1	12/13/90	K. Nguyen	Issue resolution after review.
	12/17/90	A. Brandt	
	12/20/90	M. Lang	
1.2	1/8/90	M. Lang	Editing revisions.
1.3	10/29/92	Eric Hildum	Revising friend function and type definition recommendations and requirements.
1.4	1/27/93	Lisa Luke	Revised for ISO: Added Approval sheet & Page count
1.5	4/14/95	Siamak Ashrafi	Remove C++ portion of Document

References

(1) C/C++ Coding Standards by Michael Giertych

(2) THOR Development Environment by Allen Aubuchon

(3) GNU Coding Standards

(4) VxWorks Coding Conventions

(5) The BNR UNIX Subset for Application Portability
by Jim O'Connor, David Bennett, John Hampton

1 Introduction

1.1 Purpose

Coding standards are important for several reasons. They provide higher quality code that is readable, and portable. They increase designer efficiency, and allow for tools automation. With the absence of coding standards the project is at risk of ending up with code using differing naming and header conventions, uncommented code, varying ways to implement I/O, stylistic differences, and use of non-portable libraries and system calls. In short, the result is code that is difficult to read, port, and maintain.

This document provides the rules and regulations that are recommended when writing C code by NT Mountain View designers.

1.2 Background

This document was compiled by taking the *C/C++ Coding Standard* document written by Michael Giertych, and adding concepts from the *THOR Development Environment*, *GNU Coding Standards*, and *VxWorks Coding Conventions* documents. It was then reviewed by all Mountain View SL-1 groups and Tools groups using the C language. Issues from the reviews were resolved and applied to the document.

1.3 Scope

This recommendation of a standard applies to all Mt. View designers and their managers who write code in C.

1.4 Document Issuance

All the people mentioned in the scope section above and their managers must have current copies of this manual. It is the responsibility of the SW managers to ensure that all their technical personnel have current copies of all SW Methodology documents that apply to them. This document is located in DOCTOOL via the TOOLS library.

1.5 Document Control

This document is owned and controlled by the Product Development Efficiency Engineering group. Changes to this document can only be made by one of the original authors or a designee of the Product Development Efficiency Engineering group manager.

2 Code Organization

2.1 Module Layout

A module is any unit of code that resides in a single source file. There are two base types of compilation files in C: .h (header) and .c (code) files.

Header files should be used for:

- constant and global variable declaration,
- functional unit interface specification.

Code files should be used for:

- functional unit code, and
- utility function code.

In each of the above cases, the code should be organized according to purpose. In addition, each module should provide a standard header comment, which is the part of project documentation. Refer to Chapter 5 - Documentation for further details.

2.2 Header Files

Header files (.h files) are used to provide global declarations for variables and constants, and interface definitions to objects and functional units.

> Functional units are groups of functions and data organized to provide a certain function. For example, the functions and data required to implement a protocol layer or provide a certain set of services would most likely be organized into a functional unit.

There are two types of header files and thus code in header files should be designed such that

- all declarations internal to a module are in one header file for that module, and
- all interface declarations are in a header file for that functional unit.

When these basic rules are followed the size of the header file is reasonable and manageable. A major benefit to a reasonably sized header file is twofold:

1. header files do not become excessively long, (i.e.: 20 pages), and
2. code files do not have pages of `#include` statements of header files.

2.2.1 Include Lists

Header files should be "self-contained", in that they include all other header files that they depend on. This is done by using `#include` statements. Files that a header file does not depend on should not be included. If the corresponding code file for the header needs to include a particular header file, it should do so in the code file and not its header.

When specifying the filename for an include file, do not use a directory structure path for the filename as subsequent compiles would then depend on this exact directory structure. Just the filename should be specified in an include:

```
#include  "startup.h" /* for the interface functions*/
```

2.2.2 Constant Declarations

Constants declarations should be placed in header files. These declarations should be organized according to function. For example, the `#define` statements which determine system resource allocation should not be mixed with `#define` statements which determine default screen color.

Another way of organizing constants declarations is to group them in alphabetical order. For example, instead of this:

```
#define MAX_SCREEN_ROW 25
#define MAX_SCREEN_COL 80
```

write:

```
#define MAX_SCREEN_COL 80
#define MAX_SCREEN_ROW 25
```

2.2.3 Enumerated Data Types

Enumerated data type definitions should be used rather than constant declarations for two reasons:

- the compiler will do type checking for you, so assignment errors are detected, and
- debuggers have access to the data.

Numbering of enum data types should be sequential starting from zero.

Example:

```
enum color   {mauve, emerald, turquoise};
```

2.2.4 Variable Declarations

Variable declarations within headers should be severely limited! Global variables create needless interactions. Variables can be declared static (to limit their scope) within code files or placed within objects.

If a variable must be declared global, it should be declared in a single header file for global variables (globals.h sounds good) where they can be seen easily. All reference to the global variables should be by `extern` variable declarations in either header files or code files (try to limit the scope of the declaration by moving it to the header file only if it needs to be there).

Don’t declare multiple variables in one declaration that spans lines. Start a new declaration on each line, instead.

For example, instead of this:

```
int  foo,
     bar;
```

or this:

```
int foo, bar;
```

write:

```
int  foo; /* description of variable foo */
int  bar; /* description of variable bar */
```

2.2.5 Type Definition

The `typedef` declaration is used to create new data type names. Type definitions should be used for all nontrivial type declarations; the names should be chosen to reflect the meaning of the data described by the type within the application domain. Using type definitions provides additional information to the compiler regarding the correct usage of the type, and will facilitate maintenance localizing the actual definition of the type to one location.

The format of the declaration should be:

```
typedef   int  Length;   /* description of Length */
```

2.2.6 Structure

The `struct` declaration is used to group a set of variables together into a structure. All structures should have a corresponding type definition. The format of the `struct` declaration should be:

```
struct  person  {
  char name[NAMESIZE];
  char address[ADRSIZE];
  long zipcode;
  long ss_number;
};
typedef person Person;
```

2.3 Code Files

Code files contain function implementations of a specific purpose. For example, code files which contain functions related to an OSI layer three protocol should not be mixed in with functions that provide basic I/O. In other words, functions that are unrelated to one another should not be put in the same code file. Furthermore, the following rules apply for functions within a code file.

- Local functions for a code file should be declared as static.
- All function declarations should be at the top of the code file.

- Functions should have a single entry point and a single exit point. The only exception is to have multiple exits for error conditions.
- `goto` statements are not allowed.
- Data should be handled through parameters or handles rather than global variables.

2.3.1 Include Lists

The include lists for code files should contain the header files for which the functions in the file depend. They should not include header files that the `#include` files need themselves.

2.3.2 Constants

The only constants that should appear in a code file are those which deal directly within a function's (or set of functions') scope. Thus, any constants which deal with the interface to the code file belong in a header file.

2.3.3 Variables

All variables which do not deal with the interface to the code should be declared static to limit their scope. The variables which are part of the interface belong in the header file as an extern reference.

Limit exporting of variables, data structures, and functions. Instead data should be handled through parameters or handles.

2.3.4 Macros

Macros should be used when performance matters. But a macro should not contain code that changes its control structure during run-time. One drawback of having macros is when using the debugger to debug them. Most debuggers will not handle macros well or not at all.

Avoid the use of double evaluating parameters. An example would be the evaluation of a lengthy calculation or a function call used multiple times for the same function.

2.3.5 Functions

Explicitly declare all arguments to functions, and declare the type returned by the function. Don't omit them just because they are `int`.

Each function should be preceded by a function header. See section 5.3 - Function Header for details.

The format style for a function definition should be:

```
int  print_message(max_num, min_num)
  long max_num;  /* description of max_num */
  int  min_num;  /* description of min_num */
{
  statement1;
  statement2;
} /* end of print_message */
```

An alternative format style would be to have the function type on the line preceding the function name, so that function definitions can be easily located via the Unix command `grep`. Example: `grep ^print_message *.c`

Alternative format:

```
int
print_message(max_num, min_num)
  long max_num;  /* description of max_num */
  int  min_num;  /* description of min_num */
{
  statement1;
  statement2;
} /* end of print_message */
```

3 Programming Style

This chapter defines the practices to which developers will adhere when writing C code developed at NT Mountain View.

3.1 Naming Conventions

To provide consistency, and ease of readability the following naming conventions shall be adhered to. For variables and functions two styles are recommended, Style A and Style B. Do not mix the use of Style A and Style B conventions. Projects have the discretion as to which style will be used and then should enforce the rules through code reviews.

3.1.1 Variables

Variables should be named according to the role they play in the system or code. Thus variable names such as a, b, and c, are not acceptable.

3.1.1.1 Style A

All variables shall be in lower case, and if the name contains more than one word it should be separated with underscores. All global variables shall begin with `g_ to` visually recognize globals easily.

Example:

global:	`g_max_frame_bufs`
local:	`max_frame_bufs`

3.1.1.2 Style B

All variables shall be in mixed case, where the first letter of the variable is upper case and all other characters of the word are in lowercase. If the variable name contains more than one word, each word should begin with an upper case character. No distinction is made between local and global variable names.

Example:

global:	`MaxFrameBufs`
local:	`MaxFrameBufs`

3.1.2 Function Names

3.1.2.1 Style A

All function names shall be in lower case, and if the name contains more than one word it should be separated with underscores.

Example:

function: `max_frame_bufs()`

3.1.2.2 Style B

All function names shall be in mixed case, where the first letter of the name is uppercase and all other characters of the word are in lowercase. If the function name contains more than one word, each word should begin with an uppercase character.

Example:

function: `MaxFrameBufs()`

3.1.3 Constants

All constants and `#define` statements should be in upper case. This allows easy differentiation of constants and variables. This rule applies to both Style A and Style B.

Example: `MAX_COUNT`

3.1.4 External Names

External symbols that are not documented entry points or variables for the user should have names beginning with '_'. They should also contain the chosen group name prefix, to prevent collisions with other names.

3.1.5 Macros

Macros, like `#define` constants, are always in upper case.

3.2 Code Layout

3.2.1 Function Size

For readability and maintainability reasons, the length of a procedure or function should not exceed one page or 50 lines (including braces). If you exceed this, you must have a reason.

3.2.2 Paging

Formfeed characters (^L) should be used to divide program into pages at logical places (but not within a function). These should be placed before a procedure or function header.

3.2.3 Vertical Spacing

- Blank lines make code more readable and group logically related sections of code together. Put a blank line before comment lines.
- Each statement should be placed on the separate line. The only exception is the `for` statement where the initial, conditional, and loop statements may be written on a single line:

```
for (i = 0; i < count; i++)
```

- The `if` statement is not an exception: the executed statement always goes on a separate line from the conditional expression:

```
if (i > count)
{ /* i beyond limit */
   i = count;
} /* i beyond limit */
```

```
if (i > count)
{ /* i beyond limit */
   i = count;
} /* i beyond limit */
else if (i == count)
{ /* limit reached */
   i += count;
} /* limit reached */
```

- Section braces ’{’ and ’}’ should be used for compound and singular statements. Single statements should have braces to help avoid problems if you add a statement. A concise comment statement when you open the brace and close the brace are recommended to avoid confusion with nested if statements. Either of the following two formats is acceptable:

  1.

  ```
  if (foo)
  { /* foo */
     if (bar < 0)
     { /* bar negative */
        win ();
        tally ();
     } /* bar negative */
     else if (bar > 0)
     { /* bar positive */
        lose ();
     } /* bar positive */
  } /* foo */
  ```

  2.

  ```
  if (foo) { /* foo */
     if (bar) { /*bar */
        win ();
        tally ();
     } /* bar */
     else { /* not bar */
        lose ();
     } /* not bar */
  } /* foo */
  ```

The Unix command `indent` formats C programs. To get format 1, specify option “`-br`” to `indent` and to get format 2, specify option “`-bl`”. See the Unix manual page `indent(1)` for additional information.

3.2.4 Horizontal Spacing

- Continuation lines should tabbed up with the part of the preceding line they continue, using tabs where possible to enhance readability and understanding.

  Example 1, instead of this:

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

  write:

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

  Example 2, instead of this:

  ```
  status = fooList (foo, a, b,
          c, d, e);
  ```

  write:

  ```
  status = fooList (foo, a, b,
                    c, d, e);
  ```

  Example 3, instead of this:

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

  write:

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

3.2.5 Indentation

- When indenting lines, tabs--not spaces--should be used.
- The module and function headers and the function declarations should start in column 1.
- Indent one tab stop after the curly brace ’ { ’:

  a. function declarations. Example:

  ```
  int function_name (variable_name)
     char *variable_name;
  {
     statement;
  ```

b. conditionals (see next page)

c. looping construct

d. `switch` statement (see next page)

e. structure definitions in `typedef`

f. structure declarations (see section 2.2.6)

A tab is 8 spaces. This makes tabs work well when printing and to get uniform spacing in statements when tabs are used.

- The `else` of a conditional has the same indentation as the `if`. Thus the form of the conditional is:

```
if (condition)
{ /* comment */
    statements
} /* comment */
else
{ /* comment */
    statements
} /* comment */
```

or:

```
if (condition) { /* comment */
    statements
} /* comment */
else { /* comment */
    statements
} /* comment */
```

- The general form of `switch` statement is:

```
switch (tag)
{ /* comment */
   case ’a’:
        statements
        break;

   case ’b’:
        statements
        break;

   default:
        statements
```

```
            break;
    } /* comment */
```

- Comments have the same indentation as the section of code to which they refer.
- Section braces `'{'` and `'}'` have the same indentation as the construct to which they refer.

3.2.6 Nested Logic

All nested logic is to be indented by the number of tabs equal to it's nesting level. All statements at the same nesting level shall have the same indentation. Exceptions to this rule might be continuation lines that are indented even more for readability.

3.3 Multiple Includes Collision

To avoid redeclaration errors due to including the same file twice, a `#define` statement should be used for each header to indicate that the file has been compiled into the code. `#ifdef` or `#ifndef` statements (encasing the declaration) are then used to check for the declaration before including the file.

For example, in figure A, the library file `superio.h` contains definitions for function `SuperX(int)` and three variables `x`, `y`, and `z`. Two other header files, say `header1.h` and `header2.h`, depend upon the definitions in `superio.h`. A third file, `confused.c`, depends on the definitions in both header files. If both header files are put in the same `#include` list, then there will be a compilation error (redeclaration of `x`, `y`, `z`, and function `SuperX` with parameter signature `int)`.

There are two methods of getting around this problem. One is to understand what the dependencies are for all header files in the system (unlikely without some considerable administration effort) or to use the `#define - #ifndef` macros as shown in figure B.

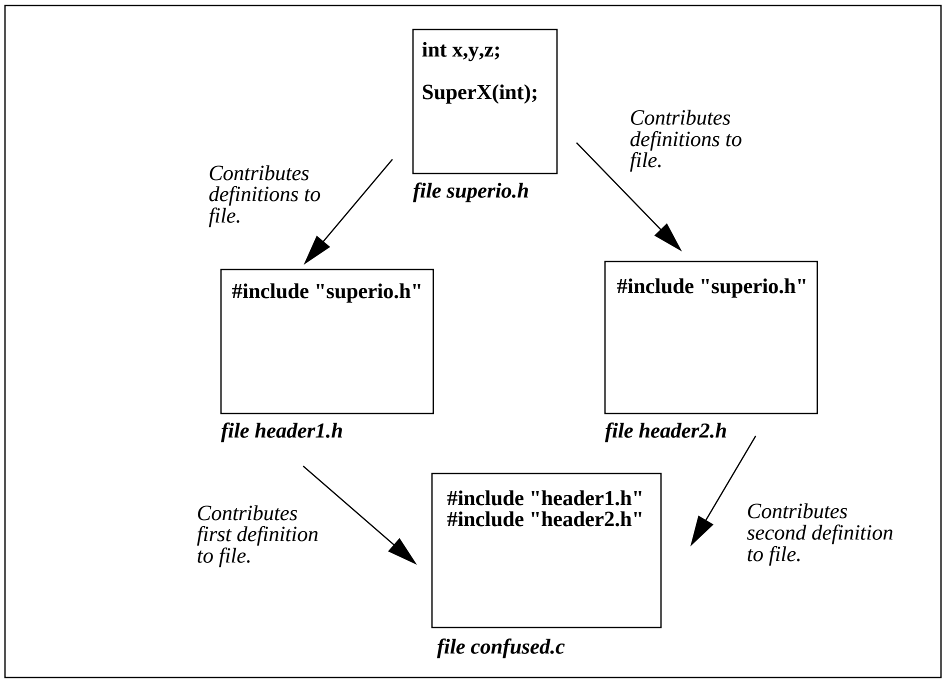

Figure A. The header dependency problem.

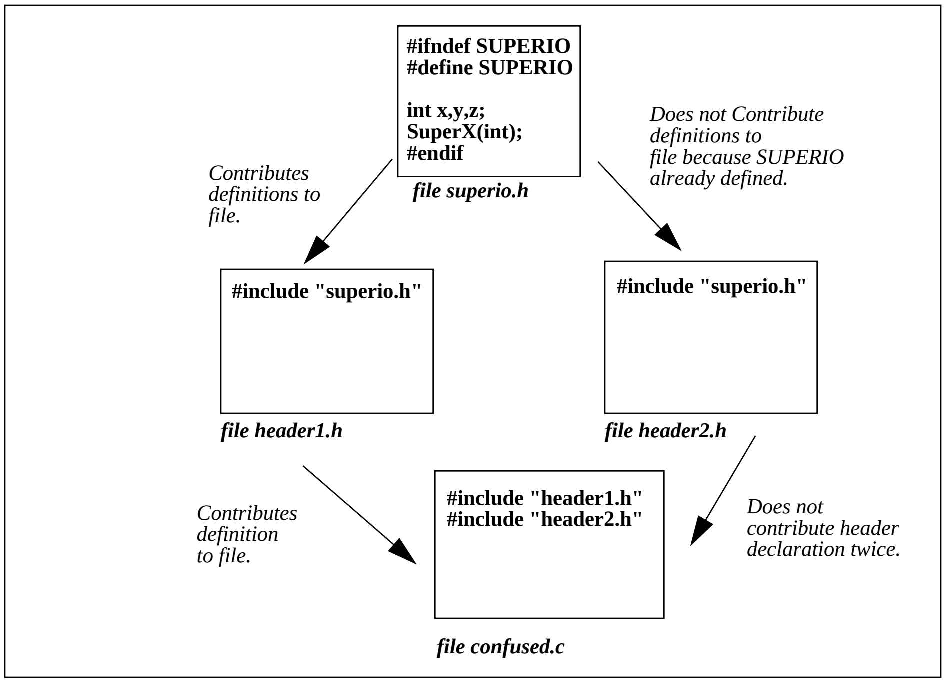

Figure B. the #define - #ifndef solution for the header dependency problem.

This method effectively removes the possibility for collisions due to multiple includes of a header file.

4 Tips and Techniques

4.1 Constant Declarations and #define Statements

When using an ANSI C compiler, constants should be defined using the `const` keyword definition as opposed to `#define`. The reason for this is that `#define` statements are evaluated before compilation time and are not seen by the symbolic debugger. Using the `const` construct assists debugging. However, the Sun compiler does not yet recognize `const`.

4.2 Variables Initializing

When static storage is to be written in during program execution, use explicit C code to initialize it. Reserve C initialized declarations for data that will not be changed.

4.3 Multiple Platforms

All projects should consider portability issues when dealing with multiple target environments (Sun, VME, Gamma) during software development and testing. The following coding techniques will help to transparently support all targets and isolate the areas of code that are target dependant:

- Any address that is target dependant should be specified with a `#define` statement. Constants should be provided for each target through `#ifdef` statements. Use of constants should be the rule, not the exception. For example:

```
#define QUEUESIZE   5       /* For all targets */
#ifdef SUN
#define IOADDRESS(&some data)
#endif
#ifdef VME
#define IOADDRESS<VME data address>
#endif
#ifdef GAMMA
#define IOADDRESS<Gamma I/O address>
#endif
```

- Any piece of data that is accessed, tested or set in a number of places or is susceptible to change should be processed through some type of data hiding mechanism. This can be through procedure calls (if performance is not an issue) or through macros. This allows the definition to change without modifying the source code, only a recompile is required. Furthermore, this change happens in only one place. For example:

```
#define READSTATUS()(StatusWord.StatusFlag)
#define ISREADY(dev)(((dev)->Status.Ready) == OK)
#define CLEARFLAG(queue)((queue)->Flag = CLEAR)
```

- Try to avoid low-level interfaces to obscure Unix data structures (such as file directories, utmp, or the layout of kernel memory), since these are less likely to be portable or work compatibly. If you need to find all the files in a directory, use `readdir(3)` or some other high-level interface.

4.4 Testing and Debugging

Building monitoring code into programs early is highly useful for testing purposes. This code can be defined for debug mode only both by conditional compilation and run-time flag(s). Multiple types of monitoring that can be turned on and off is also very useful. One possible implementation is:

Having something like this in your header file:

```
#ifdef DEBUG
#define ifdebug(what){what}
#else
#define ifdebug(what) /*{what}*/
#endif
```

and then having something like this in your code:

```
int ProcessQueue(QueueNumber)
int QueueNumber  /* Queue descriptor number*/
{
  some code
  ifdebug (printf("ProcessQueue.descriptor %d\n",
                  QueueNumber));
  ifdebug (printf("Line %d, File %s\n", __LINE__,
                  __FILE__));
  more code
}
```

In the above example, `DEBUG` is a flag you can set at compile-time to enable or disable debug statements. Note that when the C pre-processor (`cpp`) runs, it replaces the special symbols `__LINE__` and `__FILE__` with the current line number and file name.

5 Documentation

5.1 Inline documentation

- Logic Flows: The documentation that sits inside function logic is important in that it helps to describe what the code is doing at each segment. The standard approach to this type of documentation is to document the meaning of a branch in one of two ways: with a comment after a logic branch, or a block comment before the logical branch. Figure C displays this concept.

```
/* a block or several line comment */
/* on a logic step that is complex */
if ((x < y) && (y || z) || (((678 & x) * y) | t))
{ /* comment */
   statement;
   statement;
   if (x < y)
   { /* this comment when logic is simple */
      statement;
      if (y > x)
      { /* also, comment should be at same indentation */
         statement;
      } /* always have an end comment */
   } /* comment */
} /* comment */
```

Figure C. inline comments sample

Note that branch comments should start at the same indentation as the logical branches that they are documenting.

- `Switch`/`Case` Statements: When documenting `switch`/`case` statements, a brief description is suggested to describe what each `case` is doing.

  Also, when using to have fall-through `case` statements (ones which do not terminate by using a `break` statement, but continue execution into the next `case`), they should be marked clearly by a comment to avoid confusion during maintenance.

- Flow Comments: When functions are lengthy, include a few lines of comments.

- `#endif`: Every `#endif` should have a comment, except in the case of short unnested conditionals. The comment should state the ending condition along with its true or false sense. `#else` should have a comment describing the condition and sense of the code that follows. For example:

```
#ifdef foo
   ...
#endif  /* foo */
```

or:

```
#ifdef foo
   ...
#else   /* not foo */
   ...
#endif  /* not foo */
```

5.2 Module Header

The module header consists of the blocks described below. The blocks are separated by one or more blank lines and should contain no blank lines within the block. This facilitates automated processing of the header.

- **Module:**

  Consists of one line containing the module name followed by a short description. The module name *must be* the same as the module file name.

- **Modification History:**

  Each entry contains the PLS update number, date of modification, name of the programmer who made the change, and a complete description of the change.

- **General Module Description:**

  This consists of complete description of overall module purpose and function, and especially the external interface description.

The format of these blocks is shown in the following example of a module header:

```
/********************************************
* Module: setLib.h - set definitions library
*
* Modification history:
*             JX.13, 11/01/90, John Xinu
*              added set of 81 definition,
*              added set of 85 definition.
*             BD.01, 09/04/90, Bob Dobbs
*              written
* Description:
*             This module contains structures
*             definition for all sets allowed
*
********************************************/
```

Figure D.   Sample module header

5.3 Function Header

Each function should provide a header comment, which is the part of project documentation. The function header consists of the blocks described below:

- **Function:** the function name.
- **Description:** what the function does.
- **Returns:** what values are returned (through return statement or parameter list) and what the output represents.

The format of these blocks is shown in the following example of a module header:

```
/*******************************************
*
* Function   : LightPipe
* Description: Ignite tobacco in pipe bowl.
* Returns    : SMOKE for success, or
*              NO_SMOKE for failure
*
*
******************************************/
```

Figure E. Sample function header

6 Portability Issues

6.1 Reference Guide

Code written for the SL-1 product and SL-1 tools development must be portable across different versions of the UNIX operating system. BNR RTP has written a guide that details portability guidelines across six of the versions of the UNIX operating system in use throughout Bell Northern Research and Northern Telecom. Refer to this guide for details.

BNR UNIX Subset for Application Portability, Version 2.00.0, July 6, 1990

6.2 Variables

6.2.1 Max Variable Length

Each development project should investigate the maximum length of variables supported by their target linker, and publish this information to their design team.

6.2.2 Variable Size

Sometimes external requirements demand a specific length for a variable type. In this case to ensure this requirement is met and to easily change the length in the future, define a new type in the header file:

```
typedef short int2;  /* description of int2 */
```

This type can be used to declare all variables of that width and one will know that only one type definition will need to be changed to get all those variables to be the right type.

7 THOR C/SL-1 Interface Guidelines

Access to SL-1 data elements and procedures from C is fully documented in the C/SL-1 Interface Programmer’s Reference Guide. This document is available on-line in Doctool.

The title of the document is:

C/SL-1 Interface

8 THOR Rom/Ram Interworking Guidelines

Thor software must allow for high level code and rom code to:

- Invoke the same rom routine
- Pass parameters and receive results from rom routines
- Allow rom routines to be upgraded without reloading high level code

Guidelines for

- Invoking Rom Routines From Rom
- Rom Routines Parameter Passing
- Returning Results to Rom Routines
- High Level Code Calling Rom Routines
- High Level Code Parameter Passing
- Returning Results to High Level Code
- Rom Routines Stored Data

are available in the Rom/Ram Interworking document. This document is available on-line in Doctool.

The title of the document is:

Rom/Ram Interworking