RESEARCH AND DEVELOPMENT (SYSTEMS)
E-GENTING SDN. BHD.

C++ PROGRAMMING STANDARD

Fifth Draft
Jonathan Searcy
30 June 2003

CONTENTS

1. INTRODUCTION

This document describes the programming standards adopted by E-Genting Sdn. Bhd's Research and Development (Systems) Department the development of computer programs in the C++ programming language.

The programming standards have been defined from the standards used in the development of the Dynamic Reporting System (DRS) and exist principally to maintain uniformity of style in the ongoing maintenance of that system.

Standards are defined for the C++ programming language, embedded SQL within C++ language source files and shell scripts.

The author acknowledges the contribution of Lok Farn to the development of these standards. The embedded SQL programming standards are to some degree based on her document entitled *Embedded Structured Query Language (ESQL) Standard*, 18 January 2000.

2. C++ PROGRAMMING

2.1 Source File Names

The name of a source file should be the same as the name of the major component contained in the source file including any capitalisation thereof.

For example, if the major conponent of a source file is a function called '`ReadInput (...)`', then the source file should be called '`ReadInput.cpp`'.

If the major component of a source file is a main line, the source file should have the same base-name as the executable file. For example, the source file for an executable program called '`modxlt`' should be called '`modxlt.cpp`' and the source file for an executable program called '`BgtApm`' should be called '`BgtApm.cpp`'.

The type name suffix should be omitted from the file names of header files containing type declarations. For example, if the major component of a header file is '`struct MyStruct_t { /* ... */ };`', then the file name should be '`MyStruct.h`'.

Header files containing the constant values of types declared elsewhere should still be named as if the major conponent of the header file were the type name. For example if the type name '`MyStateTyp`' is declared in a global definitions file, but the values of MyStateTyp are declared in a separate header file, then the separate header file should be called '`MyState.h`'.

The capitalisation of a source file name should follow the capitalisation of its major conponent even if the major conponent of the source file is a data structure with a name that starts in lower case. For example, if the major component of a source file is an array called '`myArray[ /* ... */ ];`', then the source file should be named '`myArray.cpp`'.

2.2 General Program Layout

Each C++ source file shall be divided into the following sections:

1. title block,
2. include files and external function declarations,
3. global variable declarations,
4. function source code.

Each program section and each C++ function shall be separated from the next by a single comment line of dashes preceded and followed by blank lines.

The following sub-sections describe the layout of each section.

2.3 Program Title Block

Each C++ source file shall be headed by a title block similar to the example presented in figure 2.1.

```
// outdat.cpp - FIND OUT OF DATE OBJECT FILES
//
// USAGE
// outdat [-l libName] [srcFil]
//
// -l            indicates that the object files are in
//               a library.
// libName       is the name of a library containing the
//               object files.
// srcFil        is the source file name (stdin if omitted).
//
// SOURCE FILE FORMAT
// obj1: src11 src12 ... src1n
// obj2: src21 src22 ... src2n
//  :      :      :         :
// objm: srcm1 scrm2 ... scrmn
//
// obj.          is an object file name.
// scr.          are source file names required to compile the
//               respective object files
//
// MODULE INDEX
// NAME          CONTENTS
// main          Main line
// NextToken     Get next file name from input stream
// LastUpdate    Get last update time
// GetLib        Load ram buffer with update times
//
// MAINTENANCE HISTORY
// DATE          PROGRAMMER AND DETAILS
// 30-01-86      QJ      Original
// 03-02-86      QJ      Added qsort/bsearch of library modules
//
//----------------------------------------------------------------
```

Program Title Block
Figure 2.1

The title block shall contain the following sections:

1. title line,
2. usage,

3. miscellaneous documentation,
4. maintenance history,
5. module index.

The title line must be included in all source files. It must also be the first line in the file. The title line must contain a comment initiator ‘//’, followed by one space, the file name, a dash (‘−’), then a short one-line description of the contents of the file in capital letters.

The usage section need only be included if the source file contains a program main line (‘main’). When included it should describe the shell command line used to initiate the program and any arguments used in the command line.

Miscellaneous documentation such as ‘INPUT FILE FORMAT’ may be added where appropriate to provide functional notes on the use of the program.

All sections such as ‘USAGE’ and ‘INPUT FILE FORMAT’ are headed by a section title preceded by a blank line.

The ‘MODULE INDEX’ section should only be included if there are two or more functions in the source file or if the name of a single function is not the same as the name of the source file.

The ‘MODULE INDEX’ section consists of a list of the names of functions defined in the source file and one line descriptions of the functions. Function names begin at column 1 and function descriptions at column 17. If the source file does not contain procedure but declares variables or macros, the names of the data types or macros may be listed in the index. The program ‘modidx’ in ‘/usr/drsbin’ may be used to automatically generate the module index from function titles.

The ‘MAINTENANCE HISTORY’ section must be included in all source files. It contains a summary of the changes made to the source file. The change summaries contain the following information:

COLUMN	**DETAILS**
3	Date of the change
17	Initials of programmer who made the change
25	Summary of the changes made

The first entry in MAINTENANCE HISTORY must contain the date the program was written, the initials of the original author and the word ‘Original’.

The title block is completed by a line full of dashes (‘---’) up to column 79.

Blank lines should be included in the title blocks as indicated in the example.

2.4 Include Files and External Function Declarations

Any preprocessor statements to include header files and any external function declarations should follow the title block. They should be specified in a section on their own, preceded and followed by lines of dashes.

An example of an include file and external function declaration section is presented in figure 2.2.

```
//----------------------------------------------------------------

#include <stdio.h>                // Standard input/output
#include <sys/types.h>            // System type declarations
#include <sys/stat.h>             // File status declarations
#include <ar.h>                   // Archive declarations

int StrBiCmp (const char*, const char *);
                                  // Blank insensitive strcmp

//----------------------------------------------------------------
```

Include Files and External Function Declarations
Figure 2.2

Each include statement or function declaration should be qualified with a comment that describes the contents of the header file or nature of the function. It is acknowledged that header file comments may be missing from some older source files. Nevertheless, header file comments are to be included in all new source files.

All functions defined elsewhere but used in the source file must be declared in function prototype statements. The function prototypes may be included in header files or declared in-line in the source file. As between the two, inclusion of the function prototype statements in header files is preferred.

External variables should not be declared in this section.

2.5 Global and External Variable Declarations

Following the section for include files there may be a section for global and/or external variable declarations. This section may be partitioned into sub-sections as appropriate for the application.

An example of a global and external variable declarations section is presented below.

```
//----------------------------------------------------------------

// GLOBAL DEFINITIONS

#define MAX_SRC          200     // Maximum no of source files
#define MAX_LIB_MOD      200     // Maximum no of library modules
#define PATH_LEN         255     // Maximum path name length

//----------------------------------------------------------------

// LAST UPDATE TIME RECORD STRUCTURE

typedef struct {
        char    updNam[PATH_LEN]; // File name
        long    updTim;          // Last update time
} Upd_t;

//----------------------------------------------------------------

// GLOBAL DATA

Upd_t   source[MAX_SRC];         // Array of source files
Upd_t   library[MAX_LIB_MOD];    // Array of library modules

//----------------------------------------------------------------
```

Global and External Variable Declarations
Figure 2.3

Each section of the variable declarations must be headed by a single line title in capitals describing the section.

Declaration of the individual variables must be in accordance with sections 2.7 through 2.13.

2.6 Function Layout

Each function in a source file shall be separated by a blank line, a line of dashes, another blank line and then a title in capital letters. The format shall be in accordance with the example in figure 2.4.

```
//----------------------------------------------------------------

// MAIN LINE

int
main (
        int     argc,           // Argument count
        char    *argv[])        // Argument value pointers
{
        char    object[PATH_LEN];// Object file name
        long    objUpdTime;     // Object last update time

        // Check for command line parameters

        if (argc == 1) {
                // ...
        }
        // ...
}

//----------------------------------------------------------------
```

Function Layout
Figure 2.4

The first line of the function declaration must contain the type of the function (e.g. ‘`int`’ or ‘`void`’) and any type qualifiers (e.g. ‘`static`’). The second line must contain the name of the function and the opening parenthesis of the argument list. If the function has no arguments the closing parenthesis must also placed on the second line. If the function has arguments, they must be declared on successive lines indented one tab stop as shown in the example. The preferred indentation levels of the argument names and comments are columns 25 and 41 respectively.

Any local variable declarations should immediately follow the first opening brace.

Local variable declarations and the outermost level of the function procedure are both to be indented to the first tab stop.

Declaration of the function arguments and local variables shall be in accordance with sections 2.7 to 2.13.

2.7 Name Selection

All names used in a C++ program shall be chosen so as to assist any future maintenance programmers to rapidly recognise the nature of the object being named.

That said, there is a very real trade-off between source congestion arising from the use of long names and difficulty in understanding arising from the use of short names.

As a rule, frequently used names are easier to remember than infrequently used names. Also, frequently used names tend to create more source file congestion than infrequently used names. As such, frequently used names should be shorter than infrequently used names even if they are reduced to an acronym. For example '`mtp`' for 'master transaction processor' is quite acceptable. On the other hand, infrequently used, complex names may need to be spelled out more verbosely. For example: '`maxIssPerLocCustDay`' for maximum to be issued per location-customer-day.

2.8 Constant Names

All scalar constants, whether defined with `#define`, `const` or `enum` shall be in upper case letters. For example: `NAME_LEN` and `MAX_FILES`. When constant names consist of concatenated words, each word should be separated with underscore characters ('_') as shown in the examples.

2.9 Type Names

Type names should start with an upper-case letter and then continue in lower-case, with the first letter of each concatenated word being in upper-case. Additionally, type names should finish with one of the following suffixes:

Suffix	**Application**
`Typ`	The suffix '`Typ`' should be appended to the type names of numeric and string variables.
`_t`	The suffix '`_t`' should be appended to the names of structure, union and enumerator types that are primarily data. The '_t' suffix should be used irrespective of whether the data structures are declared with the typedef statement or with struct, union or enum tags.
`_c`	The suffix '`_c`' should be appended to the names of structure types and classes that contain significant procedural elements.

Additionally, the suffixes '_s', '_u' and '_e' may be used for structure, union and enumerator tags when the primary type definition is declared with typedef, but a tag is needed to resolve an internal reference.

Examples of type declarations are presented in figure 2.5 below.

```
//---------------------------------------------------------------

// NUMERIC AND STRING TYPES

typedef double  DateTyp;                // Date type
typedef char    NameTyp[NAME_LEN];      // Name type

//---------------------------------------------------------------

// TABLE STRUCTURE

typedef struct {
        DateTyp         tableDate;      // Date
        NameTyp         tableName;      // Name
} Table_t;

//---------------------------------------------------------------

// BALANCED TREE BRANCH UNION

union Bra_t {
        Table_t         *braTable;      // Table entry
        struct Node_t   *braNode;       // Further node
};

//---------------------------------------------------------------

// BALANCED TREE NODE

struct Node_t {
        Bra_t           *nodeBra[2];    // Branches
        short           nodeCnt;        // Branch count
        char            nodeDataFlag;   // Data node flag
};

//---------------------------------------------------------------
```

Type Names
Figure 2.5

```
//----------------------------------------------------------------

// BALANCED TREE CLASS

class BTree_c {
        Node_t          bTreeRoot;      // Root node
public:
        BTree_c ();
                                        // Constructor
        ~BTree_c ();
                                        // Destructor
        void BTreeAdd (const Table_t*);
                                        // Add row
        const Table_t *BTreeFind (DateTyp);
                                        // Find row
};

//----------------------------------------------------------------
```

Type Names
Figure 2.5

2.10 Variable Names

Variable names should start with a lower-case letter and then continue in lower-case, except for the first letter of each concatenated word which should be in upper-case. For example: ‘`procState`’ for ‘process state’.

2.11 Function Names

Function names should start with an upper-case letter and then continue in lower-case, with the first letter of each concatenated word being in upper-case. For example: ‘`TranSelExp`’ for ‘Translate Selection Expression’.

2.12 Declaration of Variables and Arrays

Variables and arrays should each be declared on a single line containing the type declaration of the variable, the name of the variable, and a comment describing the variable as indicated in figure 2.6

```
int     srcIndex;       // Index into source arrays
int     libIndex;       // Index into library array
char    libFlag;        // Library flag (1 if library)
char    libName[PATH_LEN];// Library name
FILE    *inpFp;         // Input file pointer
int     libFd;          // Library file descriptor
```

Declaration of Variables and Arrays
Figure 2.6

Consecutive variable declarations should align on tab stops at the following points wherever possible:

1. specification of the type of the variables,
2. specification of the variable names,
3. comments describing the variables.

The preferred tab stops are as shown in figure 2.7 below.

```
        int             var;            // A variable
        ^               ^               ^
        |               |               |
        9               25              41

                      Column
```

Preferred TAB Stops for Variable Declarations
Figure 2.7

Comments describing the variables should begin with a capital letter and form a clause or sentence.

2.13 Declaration of Structures

Structures should be declared as shown in figure 2.8.

```
//----------------------------------------------------------------

// FILE NAME AND LAST UPDATE TIME RECORD STRUCTURE

struct Upd_t {
        char    updNam[PATH_LEN];       // File name
        long    updTim;                 // Last update time
};

//----------------------------------------------------------------
```

Structure Declaration
Figure 2.8

Structure declarations should be preceded by a comment line describing the structure being declared.

The declaration of the components of the structure should be indented one level from the ‘typedef’ statement and should be laid out in accordance with the specification for declaration of variables and arrays in section 2.12.

In general, structure types should have short 3 to 5 letter names followed by the standard type suffix ‘_t’.

Each structure component should prefixed by the name of the structure followed by a suffix to identify the structure component. This naming convention is particularly useful where related components occur in different structures. For example path names occurring in structures ‘Upd_t’ and ‘Tbl_t’ could be called ‘updPath’ and ‘tblPath’, thereby implying the same meaning in each of the structures.

2.14 Comments

Comments should be inserted into C++ source in two ways.

In the case of declarations of variables and arrays, comments should be placed in the same line as the declaration.

In the case of structure declarations and procedure definitions, the comment should be indented to the same level as the structure declaration or procedure definition and should be preceded and followed by blank lines.

Examples of both of these types of comments are presented below.

```
int     recLen;          // Record length
int     recCnt;          // Record count
int     fd;              // File descriptor
char    buf[100];        // Record buffer

// Count the number of records.

while ((recLen = read (fd, buf, sizeof(buf))) > 0) recCnt++;
```

Example Comments
Figure 2.9

Comments should begin with a capital letter and continue in lower or upper case. Upper case may be used for emphasis where appropriate.

2.15 Indenting

The preferred indent is 8 columns. This level is preferred because it is supported by the default editor tab stops.

If the 8-column indent causes program lines to run past the 79th column, 4-column indents may be used instead. However before resorting to 4-column indents, consideration should be given to better modularisation to avoid complex, nested control statements.

If 4-column indenting is used, it should apply throughout the affected source file.

2.16 IF, WHILE and FOR Statement Layout

The preferred layouts for `if`, `while` and `for` statements are the same. Examples of the preferred layouts are presented below:

```
        if (linCnt >= MAX_LIN) break;
```

Single Conditional Statement
Figure 2.10

```
        if (
                linCnt >= MAX_LIN ||
                strcmp (tblPnt->tblPath, updPnt->updPath) == 0
        ) break;
```

Single Conditional Statement on Multiple Lines
Figure 2.11

```
        if (linCnt >= MAX_LIN) {
                fprintf (stderr, "Too many lines\n");
                exit (1);
        }
```

Multiple Conditional Statements
Figure 2.12

```
        if (
                linCnt < MAX_LIN &&
                strcmp (tblPnt->tblPath, updPnt->updPath) == 0
        ) {
                printf ("File name found\n");
                break;
        }
```

Multiple Conditional Statements on Multiple Lines
Figure 2.13

2.17 IF/ELSE Statement Layout

The preferred layouts for the various forms of the `if/else` construct are presented in figures 2.14 through 2.16.

```
        if (strcmp(argv[1], "add") == 0)        funEnum = add;
        else if (strcmp(argv[1], "chg") == 0)   funEnum = chg;
        else if (strcmp(argv[1], "del") == 0)   funEnum = del;
        else                                    Error (INF_ERR);
```

Single Line If/Else
Figure 2.14

```
if (strcmp(argv[i], "-l") == 0) {
        i++;
        strcpy (libName, argv[i++]);
        libflg = 1;
} else if (strcmp(argv[i], "-o") == 0) {
        i++;
        strcpy (outName, argv[i++]);
        outflg = 1;
} else {
        fprintf (stderr, "Invalid flag");
        exit (1);
}
```

Multiple Line If/Else
Figure 2.15

```
// Process library specification

if (strcmp(argv[i], "-l") == 0) {
        i++;
        strcpy (libName, argv[i++]);
        libflg = 1;
}

// Process output file specification

else if (strcmp(argv[i], "-o") == 0) {
        i++;
        strcpy (outName, argv[i++]);
        outflg = 1;
}

// Process invalid flag condition

else {
        fprintf (stderr, "Invalid flag");
        exit (1);
}
```

Multiple Line If/Else with Comments
Figure 2.16

2.18 DO Statement Layout

The preferred layout of the `do` statement is presented in figure 2.17.

```
// Convert integer to string

p = buf;
do {
        *p++ = i%10 + '0';
        i /= 10;
} while (i != 0);
```

Do Statement Layout
Figure 2.17

2.19 SWITCH Statement Layout

The preferred layouts of the `switch` statement are presented in figures 2.18 and 2.19.

```
// Switch on function type code

switch (funCod) {
case FUN_AB:    PrcAb ();       break;
case FUN_OT:    PrcOt ();       break;
case FUN_CP:    PrcCp ();       break;
default:        Error(INF_ERR); break;
}
```

Switch Statement Layout - Single Line Conditions
Figure 2.18

```
switch (funCod) {

case FUN_AB:
        printf ("Function AB\n");
        PrcAb ();
        break;

case FUN_OT:
        printf ("Function OT\n");
        PrcOt ();
        break;

default:
        fprintf (stderr,
                "Invalid function code\n");
        exit (1);
        break;
}
```

Switch Statement Layout - Multi-Line Conditions
Figure 2.19

To prevent running out of page width, the `case` labels are always indented to the same level as the `switch` statement.

2.20 Expression Layout

Expressions should be laid out in free format using spaces to indicate the precedence of arithmetic or logical operations.

3. EMBEDDED STRUCTURED QUERY LANGUAGE (ESQL)

3.1 Preamble

Embedded Structured Query Language (ESQL) shall be inserted into C++ programs in accordance with the layouts described in the following sub-sections.

3.2 SQL Prefix

Notwithstanding that some ESQL preprocessors may accept more brief ways of identifying ESQL statements embedded in the host language, except for `include` and `typedef` statements, all embedded SQL statements shall use the ‘`exec sql`’ prefix in accordance with the ANSI standard. This policy maximises portability of the source code between ANSI-compliant database management systems from different suppliers.

3.3 Include Statement

Header files to be included in the source stream prior to translation of the ESQL statements may be included with the `$include` statement.

`$include` statements may be intermixed with `#include` statements in the include section of the source file.

3.4 Type Definitions

Numeric and string type names that need to be used in ESQL host language variable declarations may be defined with the `$typedef` statement.

`$typedef` statements shall be laid out in the same way as C++ `typedef` statements.

3.5 Create Table Statement

Create table statements should be laid out in accordance with the example below.

```
//----------------------------------------------------------------

// SYSTEM USERS FILE

exec sql create table suf (
        sufUcn      integer not null,       // User config no
        sufDel      smallint not null,      // Deleted flag
        sufUid      char(UID_LEN) not null,  // User id
        sufEpw      char(EPW_LEN) not null,  // Encrypt passwd
        sufNam      char(SNM_LEN) not null,  // Name
        sufApc      char(APC_LEN),          // Access power
);
exec sql create unique cluster index sufUcnInd on
        suf (sufUcn);
exec sql create index sufUidDelUcnInd on
        suf (sufUid, sufDel, sufUcn);

//----------------------------------------------------------------
```

Create Table Statement
Figure 3.1

The declaration of each table in the schema should be separated by a line of dashes.

Table column names should be prefixed with the name of the table in the same way as structure component names are in the host language.

Both table and column names should conform to the capitalisation conventions for variable names in the host language.

Character string lengths should be set with `$define` symbolic constants to facilitate synchronisation of string lengths in the host language with those in the schema.

Table index names should start with the table name, followed by the names of each of the columns in the index and end with the suffix '`Ind`'.

3.6 Host Language Variable Declarations

Host language variables shall be declared, indented, between `exec sql begin declare section` and `exec sql end declare section` statements as shown in the example below. The layout requirements for the declaration of individual variables are the same as those for the host language.

```
exec sql begin declare section;
        DateTyp         txDate;         // Transaction date
        short           cnt;            // Record count
exec sql end declare section;
```

Host Language Variable Declarations
Figure 3.2

3.7 Indicator Variables

Indicator variables, where they are required, should have the same names as the primary variables that they qualify with the suffix ‘`Ind`’. Each indicator variable should be declared on the line immediately following the line on which the primary variable is declared. For example:

```
exec sql begin declare section;
        DateTyp         txDate;     // Transaction date
        short           txDateInd;  // Indicator for txDate
exec sql end declare section;
```

Indicator Variable Declaration
Figure 3.3

Indicator variables should be qualified by the standard comment ‘`// Indicator for var`’, where ‘`var`’ is the name of the primary variable.

The type of indicator variables should be ‘`short`’, not ‘`long`’ or ‘`int`’.

3.8 Singleton Select Statement

Singleton select-statements should be laid out in accordance with the following example.

```
exec sql select ntrSltNgm, ntrSltTno, ntrSltWin,
                ntrHcTno,
                ntrHcWin
        into    :sltNgm, :sltTno, :sltWin,
                :hcTno indicator :hcTnoInd,
                :hcWin indicator :hcWinInd
        from    ntr
        where   ntrTdd = :tdd;
```

Singleton Select Statement
Figure 3.4

To aid readability, the positions of line breaks in the list of database column names should correspond to the positions of line breaks in the list of host language variable names.

Although single-line select statements are scattered throughout the existing work covered by this standard, the single-line layout is no longer acceptable.

3.9 Declare Cursor Statement

Declare-cursor-statements should be laid out in accordance with the example below.

```
exec sql declare susCur cursor for
        select  susSun, susSbm,
                susExp,
                susSre,
                susCvl,
                susCpd
        from    sus
        where   susCuk = :cuk
        order   by susSun;
```

Declare Cursor Statement
Figure 3.5

The `select`, `from`, `where` and `order` clauses should all start on a new line and be aligned on the same column. The arguments of those clauses should also be aligned on the same column.

Cursor names should be qualified by the standard suffix ‘`Cur`’.

3.10 Fetch Statement

Fetch-statements should be laid out in accordance with the example below.

```
exec sql fetch  susCur
        into    :sun, :sbm,
                :exp indicator :expInd,
                :sre,
                :cvl indicator :cvlInd,
                :cpd indicator :cpdInd;
```

Fetch Statement
Figure 3.6

The division of the list of host variable names into lines should correspond to the division of the list of table column names into lines in the `declare cursor` statement.

3.11 Insert Statement

Insert-statements should be laid out in accordance with the example below.

```
exec sql insert into suf (
        sufUcn, sufDel, sufUid,
        sufEpw, sufNam, sufApc
) values (
        :ucn, 0, :uid,
        :epw, :nam, :apc
);
```

Insert Statement
Figure 3.7

As with the singleton select statement, the positions of line breaks in the list of database column names should correspond to the positions of line breaks in the list of host language variable names.

3.12 Update Statement

Update-statements should be laid out in accordance with the example below.

```
exec sql update gtr
        set     gtrGdn = :gdn,
                gtrNln = :nln indicator :nlnInd,
                gtrNad = :nad indicator :nlnInd,
                gtrTue = :tue indicator :nlnInd,
                gtrSul = :sul,
                gtrPgl = :pgl
        where   gtrFak = LCLFAK and
                gtrTcn = :tcn;
```

Update Statement
Figure 3.8

3.13 Delete Statement

Delete-statements should be laid out in accordance with the example below.

```
exec sql delete
        from    gtr
        where   gtrFak = LCLFAK and
                gtrTcn = :tcn;
```

Delete Statement
Figure 3.9

3.14 Where Clauses

Where-clauses consisting of multiple conditions joined together by the logical ‘and’ operator should be laid out in accordance with the example below.

```
where   cxlCid = :cid and
        cxlTdd <= :tdd and
        cxlTcn is not null and
        cxlCxt in (CXTPLAY, CXTSLOT);
```

Where Clause
Figure 3.10

Each condition should be listed one after another with the ‘and’ operator positioned at the end of the line.

4. SHELL PROGRAMMING

4.1 General Program Layout

Programming style is frequently neglected in the layout of shell programs as clearly demonstrated in operating system distribution media. However this is a very poor practice as frequently the first programs a new programmer is exposed to are the shell programs to compile and install the system.

Consequently the following standards shall be maintained for the layout of shell programs.

4.2 Title Block

All Shell scripts shall be headed by a title block.

The title block for a shell program is similar to that for C++. The only differences being those necessitated by the Shell comment format.

An example of a shell program title block is presented in figure 3.1.

```
# build - BUILD THE DRS SYSTEM
#
# USAGE
#
# build [shrfun] [sst] [dyd] [mtp] [tit] [log] [small] [bgt]
#
# shrfun        Build the shared function library
# sst           Build the system status tables library
# dyd           Build the dynamic display library
# mtp           Build the Master Transaction Processor
# tit           Build the Terminal Interface Task
# log           Build the Logging Task
# small         Build the small model programs
# bgt           Build the background overlay programs
#
# MAINTENANCE HISTORY
# DATE          PROGRAMMER AND DETAILS
# 04-10-85      JS      Original
#
#----------------------------------------------------------------
```

Shell Program Title Block
Figure 4.1

4.3 Procedure Layout

In general shell procedure is to be indented to indicate the scope of conditional statements in the same manner as C++ source.

Examples of the layouts for the various conditional constructs for the shell are presented in figures 3.2 through 3.7.

```
for A in $@ do
        echo $A
        cc $A.c -o $A.o
done
```

For Statement Layout
Figure 4.2

```
case $A in

# Make the shared function library

shrfun) make -f shrfun.m shrfun.d
        cat shrfun.m shrfun.depend > makefile
        make
        ;;

# Make the master transaction processor

mtp)    make -f mtp.m mtp.d
        cat mtp.m mtp.depend > makefile
        make
        ;;

# Make the terminal interface task

tit)    make -f tit.m tit.d
        cat tit.m tit.depend > makefile
        make
        ;;

# Check for invalid argument

*)      echo "Invalid argument"
        exit 1
        ;;
esac
```

Case Statement Layout
Figure 4.3

```
if test $? != 0; then exit 1; fi
```

If Statement Layout - Single Line
Figure 4.4

```
if test -f mtp_ab.c; then
        cc -c -O -Ml -NT MTP mtp_ab.c
        ar r mtp.a mtp_ab.o
        rm mtp_ab.o
fi
```

If Statement Layout - Multiple Line
Figure 4.5

```
if test -f mtp_ab.o; then
        echo "Object file exists"
else
        echo "Object file does not exist"
fi
```

If/Else Statement Layout
Figure 4.6

```
while read A; do
        drs get $A.c
        cc -c -O $A.c
        ar r drs.a $A.o
        rm $A.o $A.c
done
```

While Statement Layout
Figure 4.7

5. PLAGIARISM

Plagiarism is the copying or use of another author's work without recognition. It is fraud, it is a crime.

In academic circles it is generally punished by immediate and irrevocable dismissal.

Where one programmer uses the work of another as the basis for a new program the original author must be acknowledged. The acknowledgement may be either a note in the title block or a comment in the body of the program as demonstrated below:

```
MAINTENANCE HISTORY
DATE            PROGRAMMER AND DETAILS
26-08-94        JS      Original, from Peter Main,
                        tulf.s, 30-06-89
```

Acknowledgement of Original Author
in Title Block
Figure 5.1

```
        // Open terminal device then wait for
        // buffers to become empty by closing the
        // device and re-opening it.  Adapted
        // from Jonathan Searcy, kit.c, 28-09-85

        if ((devFd = open (devName, 2)) == -1)
                err_sys ("kit: open tty 1");
        if (close (devFd) == -1)
                err_sys ("kit: close tty 1");
        if ((devFd = open (devName, 2)) == -1)
                err_sys ("kit: open tty 2");
```

Acknowledgement of Original Author
in Body of Program
Figure 5.1

It is not mandatory to acknowledge the original author when copying your own work.