Programming Practices And Standards

1-1-1994

Michael A. Craft
Institute for Simulation and Training

Programming Practices and Standards

Third Edition
First Revision

December 21, 1994

Prepared by
Michael A. Craft

Reviewed by
Scott H. Smith

First Edition July 31, 1991
Second Edition November 11, 1992
Third Edition September 14, 1992

Printed on December 21, 1994
Institute for Simulation and Training

Programming Practices and Standards

Third Edition
First Revision

December 21, 1994

Prepared by
Michael A. Craft

Reviewed by
Scott H. Smith

First Edition July 31, 1991
Second Edition November 11, 1992
Third Edition September 14, 1992

Printed on December 21, 1994
# Table of Contents

1. INTRODUCTION TO THE THIRD EDITION ................................................................. 2
   1.1 MOTIVATION ............................................................................................................. 2
   1.2 TERMINOLOGY ......................................................................................................... 3
   1.3 EXAMPLES USED IN THIS TEXT ............................................................................... 3
   1.4 CHANGES FROM PRIOR EDITIONS ........................................................................ 3

2. OVERALL SYSTEM DESIGN .......................................................................................... 5
   2.1 FILES TYPES ........................................................................................................... 5
   2.2 CODE ORGANIZATION BY FILE ........................................................................... 5

3. COMMENT BASICS ....................................................................................................... 6

4. FILE LAYOUT ............................................................................................................... 8

5. FILE LAYOUT (SECTION CONTENTS) .......................................................................... 10
   5.1 HEADER FILES ....................................................................................................... 12

6. OTHER POINTS ............................................................................................................ 19
   6.1 INCLUDE ORDER ..................................................................................................... 19
   6.2 FILE LENGTH ......................................................................................................... 20
   6.3 ARITHMETIC COMPARISONS AND DEMORGAN'S LAWS .................................... 20
   6.4 ASSERTIONS ............................................................................................................ 21
   6.5 CODE FACTORING ................................................................................................. 21
   6.6 BOOLEAN EXPRESSIONS ......................................................................................... 22
   6.7 TYPE CASTING ....................................................................................................... 22
   6.8 COMPILATION ERRORS AND WARNINGS ......................................................... 22
1. Introduction to the Third Edition

It may not have been made sufficiently plain in prior editions, so let it be said up front: this text outlines the programming standards used by IST personal working on IST's Testbed. The amount of latitude allowed is small. Prior versions of the standard attempted to allow flexibility, but the privilege is too often abused.

In an attempt to make this document more accessible, the emphasis has changed to illustration (examples). As far as possible issues are illustrated without further explanation in the text. It is hoped this will make the standards easier to digest. Pay attention to everything: the use of white space, the use of case in identifiers, the contents of standard comment blocks, etc.

Important points in the text sections are often emphasized with the use of an arrow, especially those points that may be difficult to illustrate, for example:

=> Programmers may be asked to correct problems they did not introduce as part of the reviewing process.

It would seem that some elements of the standards (perhaps alignment of comments and the use of case) could be automated. This is indeed true; Ada implementations come with formatters that take care of many mechanical issues. We have solicited formatters and evaluated a few, but none have added enough value to make them worthwhile. It is practical to produce a formatter in house and someday we may do so, but the bulk of issues covered here would be difficult to automate.

1.1 Motivation

A customer has a right to expect quality code in a uniform format. It is not reasonable to expect a client (or a new staff member at IST) to deal with individual programmer's eccentricities, much less poorly laid out or designed code. The size and complexity of the IST Testbed and the number of people working with it make a standardized approach essential. These coding standards address such issues in some detail.

All procedures and standards in this document are subject to change. Such changes will be reflected in the electronic version of this document which is kept with the librarian procedure documentation (refer to section 1.4).
1.2 Terminology

The IST Testbed (usually referred to as "the Testbed") is the body of software consisting of (at least) the Simulator (DIS and SIMNET), the Operator Interface (the OI), the TDB Conversion program, the Logger, the Eagle project, and the Testbed utilities. The various Testbed executables are sometimes referred to as the Testbed “products” or “applications.”

The person in charge of the Testbed libraries has the title "Software Librarian," usually referred to as "the Librarian." The Librarian’s duties and procedures are not covered here except by reference to the Librarian’s documentation.

1.3 Examples Used in this Text

As far as possible examples are from the Testbed. In most cases the code shown is incomplete to conserve space.

In many cases repairs had to be made to extracted code because of coding violations: the Testbed is far from perfect today. For this reason, questions about standards should be answered from this document, not from sample Testbed code.

Not every detail can be pointed out, but use the examples as a guide: pay attention to spacing and style. Some points are difficult to illustrate and are covered beginning in section 6.

1.4 Changes from Prior Editions

Unlike prior editions this paper does not cover software integration. For an up to date description of integration procedures see the system librarian. At the time of this writing the procedures are described in the various documents under delta support (DIS_SERVER\DELTA\TESTBED\SRC\DOCS\PROCEDUR).

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stand3.doc</td>
<td>This document (SW Standards Version 3).</td>
</tr>
<tr>
<td>Synch.txt</td>
<td>Explanation and directions for synchronizing a local copy of a project.</td>
</tr>
<tr>
<td>Delta.txt</td>
<td>Procedure for using Delta Software Control.</td>
</tr>
<tr>
<td>Ckin_out.txt</td>
<td>Procedure for checking files in and out</td>
</tr>
<tr>
<td>Bug_rpt.txt</td>
<td>Procedure for reporting bugs</td>
</tr>
</tbody>
</table>
Earlier editions of this document defined a “master to-do list.” Individual projects are now too diverse to maintain their goals on a single list, although the need for overall changes to the Testbed still arises. Such global changes, and errors in the Testbed that do not fall in the purview of the project which notes the problems, are handled through the system Librarian. Refer to the Librarian’s procedures for more information.

The whole flavor of this document has been turned to brief descriptions and examples. Some valuable code fragments have been, reluctantly, discarded in the process. The discussion of code complexity (and how it is hidden by non-structured constructs) is missing, but is available through the earlier editions.
2. Overall system design

Our goal is to design well-structured code that is easy to understand and modify, and which provides a solid platform on which to base further work. This requires a well thought out and complete design before coding begins.

The system should not be considered a large, monolithic, single entity, but rather should be considered a group of interacting modules (or objects).

Modules (objects) should be self-contained, and should rely as little as possible on other modules. The use of global data is prohibited; the interface to a module is achieved through its functions.

2.1 File types

Files fall into two categories, “headers” and “source.” The header files have a “.h” extension whereas source files are compilable files (and so this excludes makefiles) which do not have a “.h” extension; the usual extension for source files is “.cpp” although exceptions exist.

A module consists of an interface header file with a .h extension and one or more source files. If there is more than one source file in a module, a local header file may be required. Such local headers are kept in “include\local” and use names beginning with “loc_” (this may seem redundant, but this identifies the locals when included in source files). Local headers are only included in the module for which they are defined: they are for intra-module binding. Non-local headers (inter-module binding) are kept in “include\global.”

The requirements for the two file types are different and their respective organizations are covered by this document.

2.2 Code Organization by File

The general design rule is to not combine logically independent software in the same file. If a set of routines is completely subservient to another set, by all means keep them together. But, if they are logically independent, separate them.
3. Comment Basics

#include __VERSION
#define __VERSION

/**/
* Definition of the version number, displayed at system start up. *
* *
* Responsibility: Doug Wood
***************************************************************************/

/*** INCLUDES ***/

/*** CONSTANTS ***/
#define DATE "IST08/23/94"
#define VERSION "7.05.08"

/*** TYPES ***/

/*** FUNCTIONS AND PROTOTYPES ***/

#define

/*** END OF FILE ***/

Block comments RHS is in column 77. Blocks are only used for file headers and function headers.

Don't start comments with "This file" or "This function." Just say what is done, no self-reference is needed.

All header files contain these comments.

ANSI Comments only.
Gather executive profile information.

Responsibility: M. A. Craft

Files preceding this required pragma (only used in sources) are listed in the text.

These comments (and the end of file comment) appear in every non-header source file.

Align comments locally.

Line comments are indented at the same nesting level as the code being described.

Appears in every file.
4. File Layout

#ifndef _LOC_TERR
#define _LOC_TERR

/****************************************************************************
* These definitions are used within the terrain module to perform:
* - (FileIO) reading database file and closing file
* - Current database specific operations
* 
* Responsibility: Doug Wood
****************************************************************************/

/***
*** INCLUDES ***/

#include "hx.h"
#include "tdb_type.h"
#include "terrain.h"

/***
*** CONSTANTS ***/

/***
*** TYPES ***/

/* The DB map is followed by the patch indices and the patch elevation */
/* minima/maxima. The patch minima/maxima are organized as patch guards. */
/* These guards allow intervisibility to avoid some processing. This data is read */
/* into memory if intervisibility is in use. The hope is that we will avoid reading */
/* the patch altogether, saving an expensive disk access. */

A block comment would be inappropriate here as this is not a file or function header comment. Nonetheless, the comment edges are aligned. A single space separates the lead asterisk and the first character of the verbiage.
typedef struct
{
    float min_z_left_edge; /* Min z along left edge */
    float min_z_bottom_edge; /* Min z along bottom edge */
    float max_z_polygon; /* Max polygonal elevation */
    float max_z_object; /* Tallest object */
    float max_z_trees; /* Tallest tree on the patch */
    float max_z_treelines; /* Tallest treeline on patch */
    float max_z_canopies; /* Max z in a tree canopy */
} PATCH_GUARD;

/**/ FUNCTIONS AND PROTOTYPES /**/

***************************************************************************
* Using the values in the DB_HEADER, calculate the indices of the 8 patches *
* bordering the patch indicated by "center_index".
**************************************************************************/

void GetBorderIndices
(
    long center_index, /* Index for the center patch of the new region. */
    long *this_index /* Array to store the calculated patch index values. */
);

***************************************************************************
* Dereference a patch handle. Bail out of the system if anything goes wrong. *
* This routine should only be called via the macro getPatchHeader *
* *
* Returns: the patch header pointer corresponding to the patch handle. *
**************************************************************************/

PATCH_HEADER *GetPatchHeaderDebug
(
    HX_HND handle, /* The patch handle to be dereferenced */
    char *file, /* File from which the call was made */
    int line /* Line on which the call was made */
);

#endif

/*** END OF FILE ***/
5. File Layout (Section Contents)

```
/****************************************************************************
* Memory Debugging Support
* Responsibility: M.A.Craft
****************************************************************************/

/*** INCLUDES **/
#include <stdio.h>
#include <stdlib.h>
#pragma hdrstop
#include <alloc.h>
#include <assert.h>

/***
CONSTANTS
***/
#define LIST_SIZE 2048
#define CHECK_SIZE sizeof(char)

>Returns pointers must be aligned on valid boundaries.
#define ALIGNMENT_SIZE 4     /* Valid byte boundary size. */
#define CHECK_BIT_OFFSET 24   /* Bit shift to add first check byte. */

#define BEGINNING_CHECK '@'  /* Set values to be used for boundary */
#define ENDING_CHECK '! '    /* checks. */

/*** TYPES **/

>Returns to determine if memory allocation should exit on failure. */
enum FAIL_ENUM
{
    CONTINUE,
    QUIT
};
```

Contains #define'd constants and anonymous enum's (not macros). Named constants are used in place of raw numbers unless meaning is obvious (e.g., 0). The Testbed does not use "const" globals.

Values to determine if memory allocation should exit on failure. */

Use enumerations whenever appropriate: typically classification values, especially when the values are unimportant or when the values are consecutive.
typedef struct
{
    void  *address; /* Address of memory allocated. */
    char *file;    /* File name where allocation occurred. */
    int    line;  /* Line number where allocation occurred. */
    UINT   amt;   /* Amount of memory allocated. */
} MALLOC_LIST_ENTRY;

/*** STATIC DATA ***/
static int debugging = TRUE, /* Is debugging on? */
               first_free = 0;     /* First free malloc list index. */

static MALLOC_LIST_ENTRY *malloc_list = NULL;

/*** LOCAL PROTOTYPES ***/
void DumpMalloc (void);

/*** FUNCTIONS ***/

/************************* **************************************************
* Begin tracking pointers to malloc'd memory
************************* **************************************************
void EnableDebugMemory(void)
{
    InitializeDebugMemory();
    use_debug_mem = TRUE;
}

/*** END OF FILE ***/
5.1 Header files

#include "dynamics.h"

/**** INCLUDES ****/

Header files never contain code or variable declarations.

Header's block comments state the purpose of the module and who is the main source for information about the module. Information pertinent to the use of the module but not specific to any function is also here.

/****************************************************************************
* Module supports:
* 1. Checking for object collisions
* 2. ...
* *
* Responsibility: Sumeet Rajput
****************************************************************************/

/**** INCLUDES ****/

#include "dynamics.h"

/**** CONSTANTS ****/

/**/ Headers are the interface spec's for modules. They contain all info needed for programmers to use the module resources.

Anonymous enums appear in the CONSTANTS section.

enum
{
    SPEED_TABLE_DI = 0,
    SPEED_TABLE_VEHICLE,
    SPEED_TABLE_LANDED_FIXEDWING
};

/* We handle speed calculations for three types of entities. */
/* These values are used to index into the speed table. */
#if 0
/** For demonstration only. */
enum
{
    ACC_TABLE_DI = 0,
    ACC_TABLE_VEHICLE,
    ACC_TABLE_LANDED_FIXEDWING
};
#endif

/*** TYPES/***

/*** FUNCTIONS AND PROTOTYPES/***

****************************************************************************
* Determines if the vehicle identified by e_cb has collided with an object or
* another vehicle. Return indicates whether collision occurred or not. If the
* return indicates a collision with a vehicle, the ID of that vehicle is supplied in
* thing_hit. If a collision with a terrain object INVALID_ID is returned in
* thing_hit.
*
* Return: did a collision occur?
****************************************************************************/

BOOL CheckCollision
(
    ENTITY_CB *e_cb,    /* pointer to Entity Control Block */
    VEHICLE_ID *thing_hit /* address of a VehicleID structure */
);

/*** END OF FILE/***

#endif

All functions within a module used in other modules MUST be prototyped in a header file. The prototype includes the return type (possibly void), the function name, and the parameter list including parameter names.

Comments do not contain redundant information (such as the function name). State the use or implementation directly without self references (avoid saying "this function").
Functions are preceded by a block comment. For source files, discuss implementation, particularly if it is unusual or complicated; information in header files is not duplicated here. For static functions (no prototype) both use and implementation may be discussed.

```
/* Process a list of information to be stored. The list contains sub-lists of numeric * 
data consisting of a range and the damage probabilities for that range. The * 
* number of sub-lists is counted and stored, ... */
****************************************************************************

static int StoreIFInfo
{
    CHAIN_HEAD *head,
    int *veh_list,
    int vehicle_number,
    int *mun_list,
    int munition_number,
    char **error_string
    /* pass error message */

    LIST_NODE *old_node; /* save old current node */
    int readin_ok = TRUE; /* assume all will go well. */

    ... /* A reasonable notion of what is going on should be available through comments alone. They do not echo the code, they explain it. */

    /* Count the number of elements in the list */
    while (current_node = (LIST_NODE *)GetNextNode(current_node))
        num_elements++;

    /* Malloc IF damage info space for these vehicle and munition types */
    for (vtmp=0; vtmp<vehicle_number; vtmp++)
        for (mtmp=0; mtmp<munition_number; mtmp++)
        {
            if_table[veh_list[vtmp]].detonator[mun_list[mtmp]].damage_list =
                (IF_DAMAGE_DISTANCE *)
                calloc(num_elements, sizeof(IF_DAMAGE_DISTANCE));
        }

    return readin_ok;
}
```
switch (add_data->entity_type) {
    case EGL_MGR_TYPE:
        if (!created = CreateDynamicCB(&new_entity,...)) {
            TraceOutput(ERROR,"Unable to create CGF Eagle Manager.
            a\n");
            break;
        }
    default:
        created = FALSE;
        TraceOutput(ERROR,"Can't create unknown Eagle entity type
        a\n");
        break;
}

if (created) {
    /* The entity was successfully created, send an initialize */
    /* message to the new entity. */
    MsgBuildAndSend(&new_entity,
        INITIALIZE,
        sizeof(VEHICLE_ID),
        &add_data->immediate_sup,
        NO_DELAY,
        RESPONSE_NOT_NEEDED,
        IncGcbSeq());
}
/* The Aggregation FSM of the Eagle Manager. */

* Responsibility: David Stober

****************************************************************************/

/***
** INCLUDES 
*** /

Identifiers (types, structures, constants) of only local interest should only appear in the file that uses them. If identifiers are module bound, put them in the module's local header.

#include "core.h"
#pragma hdrstop
#include <assert.h>
#include "imn_mgr.h"

/***
** CONSTANTS 
*** /

/* The status of an aggregation request in the aggregation list. */
enum
{
    AGG_WAITING, /* Aggregation waiting for the unit's disagg to complete */
    AGG_READY,   /* Ready for aggregation */
    AGG_ACTIVE,  /* Aggregation currently running for this unit. */
};

#define MAX_EM_AGG_LIST 20 /* Phoney define for demo only */

/***
** TYPES 
*** /

typedef struct
{
    CHAIN_LINK link; /* The links to the nodes in the list. */
    MESSAGE *msg;   /* The aggregate request. */
    int status;    /* Status of the element in the aggregation list. */
} AGG_LIST_ELEMENT;

/***
** STATIC DATA 
*** /

static CHAIN_HEAD em_agg_list; /* The Eagle Manager's aggregation list. */

/***
** LOCAL PROTOTYPES 
***/
static void RemoveC2Node(FSM_RECORD *fsm);

/***
 *** FUNCTIONS ***/

#define EMAggFSMData(fsm)((EM_AGG_FSM_DATA *)(EglFSMData(fsm)->data))

#define EMAggFSMData(fsm)((EM_AGG_FSM_DATA *)(EglFSMData(fsm)->data))

/* Retreive the Eagle Manager's AGG FSM data from an FSM record. 
 * fsm should be the FSM record of the Eagle Manager's Aggregation FSM. */
*
**************************************************************************
* The start state of the Eagle Manager Aggregation FSM.
* Determine the root C2 Node of the disaggregated unit and sends it an
* aggregation message.
**************************************************************************

void EglMgrAgg(FSM_RECORD *fsm)
{
    CHAIN_LINK *c2_list = CLink(&EglMgr(fsm->cb)->c2_node_list);

    TraceOutput(DEMO,"Eagle Manager Aggregation Beginning\n");

    /* Search the list of C2 nodes for this unit. */
    while (found && (c2_list = GetNextNode(c2_list)))
    {
        found = ((C2_NODE_ENTRY *)c2_list)->unit_id == unit_id;
    }

    /* arch the list of C2 nodes for this unit. */
    while (found && (c2_list = GetNextNode(c2_list)))
    { found = ((C2_NODE_ENTRY *)c2_list)->unit_id == unit_id; }

    A blank line precedes and follows control constructs (if, switch, while, etc.) so the construct stands out from surrounding code.

Variable names are to be meaningful.
Control statements are not functions: leave a space between the control word (if, while, switch, etc.) and the parenthesis. No space is required after the opening parenthesis or before the closing parenthesis.

if (!found) {
    TraceOutput(ERROR, "Unit %lu not in list of units that it is maintaining.", unit_id);
    NewState(fsm, DoneAgg, STD_DELAY);
} else {
    /* Found the C2 node for this unit, so remember its ID */
    /* and send it a message to aggregate */
    EglFSMDatad(fsm)->data = malloc(sizeof(EM_AGG_FSM_DATA));

    /* Release all the manned simulators associated with this unit. */
    MsgBuildAndSend (&(EglMgr(fsm->cb))->man_mgr_id),
    RELEASE_SIMULATORS,
    sizeof(UNIT_ID),
    &unit_id,
    NO_DELAY,
    RESPONSE_NOT_NEEDED,
    0);

    fsm->current_state = RemoveC2Node;
}

/*** END OF FILE ***/
6. Other Points

The examples cannot capture everything. Here are other important items.

⇒ Use standard functions, don’t write your own equivalents.

⇒ (char *) is not used as a generic pointer type, use (void *).

⇒ Inline assembly code is not permitted.

⇒ The comma operator is disallowed

⇒ Floating point comparisons for equality are suspect: use Zero, Tiny, Small or Close.

⇒ Remove tabs before integration (a utility is available from the Librarian).

⇒ Non-structured constructs are disallowed. These include goto’s, continue’s, break’s (with the exception of breaks in cases), and returns (except at the end of a function).

⇒ As functions are developed, they are given a scope as local as makes sense. Adding function prototypes to a global module header is a serious step.

6.1 Include Order

The #include order before the pragma hdrstart is:

<table>
<thead>
<tr>
<th>CORE</th>
<th>SIM</th>
<th>OI</th>
<th>EAGLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;stdlib.h&gt;</td>
<td>&lt;stdlib.h&gt;</td>
<td>&lt;graphics.h&gt;</td>
<td>&quot;core.h&quot;</td>
</tr>
<tr>
<td>&lt;string.h&gt;</td>
<td>&lt;string.h&gt;</td>
<td>&lt;stdlib.h&gt;</td>
<td>&quot;core.h&quot;</td>
</tr>
<tr>
<td>&quot;core.h&quot;</td>
<td>&quot;core.h&quot;</td>
<td>&lt;string.h&gt;</td>
<td>&quot;core.h&quot;</td>
</tr>
<tr>
<td>&quot;coreutil.h&quot;</td>
<td>&quot;coreutil.h&quot;</td>
<td>&quot;operator.h&quot;</td>
<td>&quot;map.h&quot;</td>
</tr>
<tr>
<td>&quot;exec.h&quot;</td>
<td>&quot;ent_mgr.h&quot;</td>
<td>&quot;fsm.h&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;simaddr.h&quot;</td>
<td>&quot;fsm.h&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The files above the "#pragma hdrstop" must appear as indicated; any of these files not required can be omitted, but other header files should not be added above the #pragma.
If none are required, the #pragma must appear anyway (with nothing above it). After the #pragma, the ordering (if possible) is system header files in alphabetical order followed by our header files in alphabetical order.

6.2 File Length

Source files may not exceed 1200 lines. Any file longer than 1000 lines is considered too long, and steps should be taken to break it into smaller files. Removing white space and documentation are not legitimate methods of reducing file size. At the other extreme unnecessary file splits are discouraged and very small files should be combined with logically related files whenever appropriate.

6.3 Arithmetic comparisons and DeMorgan's Laws

When checking that a value is in, or out, of a range, a number line ordering is recommended:
if (A <= x && x <= B)
not, for example,
if (x >= B && x <= A).
If x is to be out of the span, write it as "if (x < A || B < x)."

Many very complex tests can be simplified with the application of DeMorgan’s Laws:

(Not (A and B)) ⇔ ((Not A) or (Not B))
(Not (A or B)) ⇔ ((Not A) and (Not B))

6.4 Assertions

Occasionally a programmer will recognize a condition would cause a problem, but the condition should not occur.

If in doubt, include an assertion. For example, if dt should never be zero, but its computation is complex (or depends on parameters), the developer should consider including "assert(!zero(dt));" before dividing by dt.

6.5 Code factoring

Using an editor to copy code is seductive, but usually inappropriate. Rather than copy code, create a function.
6.6 **Boolean expressions**

Booleans are not tested simply to set booleans. For example,

```c
bool = TRUE;
if (a < b)
    bool = FALSE;
```

is written `bool = b <= a`.

6.7 **Type casting**

Type casting is done only when necessary. Casting an identifier to its own type is not done. Casting is not used when a formal parameter is `(void *)` and the actual parameter is a pointer.

6.8 **Compilation Errors and Warnings**

The Librarian will refuse integration of any file which fails to compile and may refuse any file which generates a warning. Pragmas to disable warnings of unused parameters may be allowed on a case by case basis.