# ACIS Science Instrument Software Detailed Design Specification (Code-To)

**MIT Center for Space Research**

36-53200 Rev. 01

May 22, 1995

# 1.0 Introduction (36-53201 01)

The AXAF-I CCD Imaging Spectrometer (ACIS) Science Instrument Software (SIS) is being developed by the Massachusetts Institute of Technology, Center for Space Research (MIT-CSR) as part of the ACIS Digital Processor Assembly (DPA). The DPA resides on-board the Advanced X-ray Astrophysics Facility - Imaging (AXAF-I). The DPA Science Instrument Software is responsible for acquiring and processing image data from the ACIS CCD Imaging Spectrometer and transferring the processed data to the AXAF-I Command and Telemetry Unit (CTU), which is then responsible for sending the information to the ground.

## 1.1 Purpose

The ACIS Science Instrument Software Detailed Design Specification (Code-To) describes the design of the instrument software in sufficient detail to permit code development.

## 1.2 Scope

This document applies to the detailed design of the ACIS DPA Science Instrument Software. It does not provide information for the Ground Support Software (GSS), which is maintained separately as part of the Electronic Ground Support Equipment (EGSE).

This document supplies information applicable to SDM03 from the original contract, and to DM09 from MM8075.1.

By mutual agreement, MSFC Software Management and Development Requirements Manual MM8075.1, which supersedes MA-001-006-2H, forms the basis for this document.

## 1.3 References

This specification relies on a set of existing documentation. The following table lists these documents.

**TABLE 1. Reference Documents**

| Part Number | Version | Title |
|---|---|---|
| MSFC MM 8075.1 | January 22, 1991 | MSFC Software Management and Development Requirements Manual |
| MIT-CSR 36-01103 | B | ACIS Science Instrument Software Requirements Specification |
| MIT-CSR 36-01502 | 04 | ACIS Technical Analyses and Models: ACIS Hardware Specification and System Description |
| NU910701 | 1991 | Nucleus RTX Reference Manual from Accelerated Technology, Inc. |
| NU910702 | 1991 | Nucleus RTX Internals Manual from Accelerated Technology, Inc. |
| ISBN 0-8053-5340-2 | 1994 | Object-Oriented Analysis and Design with Applications, Second Edition by Grady Booch, Benjamin/Cummings |
| NASA Reference Publication, 1319 | September, 1993 | Mongoose ASIC Microcontroller Programming Guide, Brian S. Smith, GSFC |
| ISBN 1-55860-297-6 | 1994 | MIPS Programmer's Handbook by Erin Farquahar and Philip Bunce, Morgan Kaufman Publishers |
| ISBN 0-13-584749-4 | 1989 | MIPS RISC Architecture, by Gerry Kane, Prentice Hall |
| MIT 36-10410 | TBD | ACIS Instrument Protocol and Command List |
| MIT 36-02205 | A | DPA/DEA Interface Control Document |

# 2.0 Assumptions and Conventions

## 2.1 Audience

This document assumes that readers will be familiar with the ACIS Contract End Item Specification, and the ACIS Science Instrument Software Requirements Specification.

## 2.2 Portability

This document assumes that there are no hardware nor operating system portability requirements on the instrument software design or implementation.

## 2.3 Implementation Language

This document assumes that the Back End Processor software design is implemented in C++, and that the Front End Processor design is implemented in C. Unless otherwise specified, all data types and example code shown in this document use C++ notation.

## 2.4 Compiler Selection

This design assumes that the GNU C++ compiler, running on an DECstation 3000 or 5000, is used to compile the flight version of this software. Other compilers may be used for unit and integration testing portions of the software, but there may be parts of the software which are compiler specific.

## 2.5 Graphic Notation

Unless otherwise specified, diagrams and detailed class descriptions presented in this document use Booch Notation, as described in "Object-Oriented Analysis and Design with Applications," by Grady Booch, 1994.

Figure 1 illustrates the icons and associations this document uses to illustrate the relationships between class and structure definitions.

FIGURE 1. Class Diagram Icons

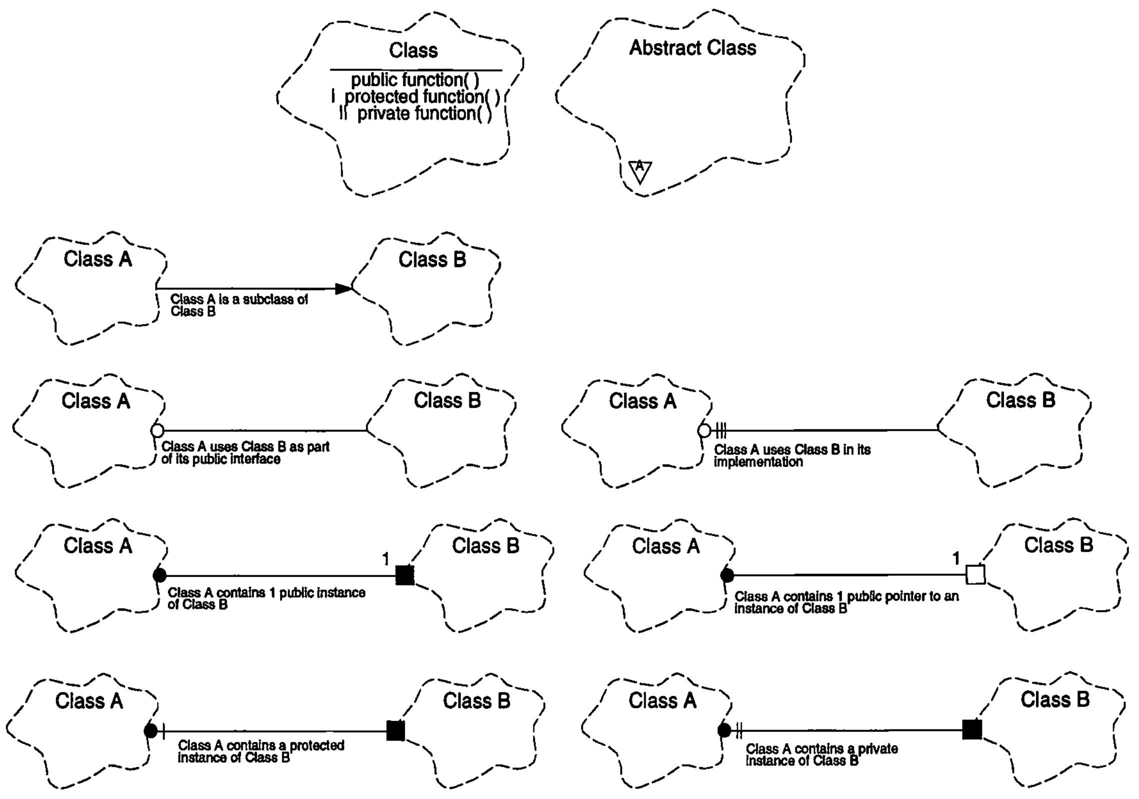
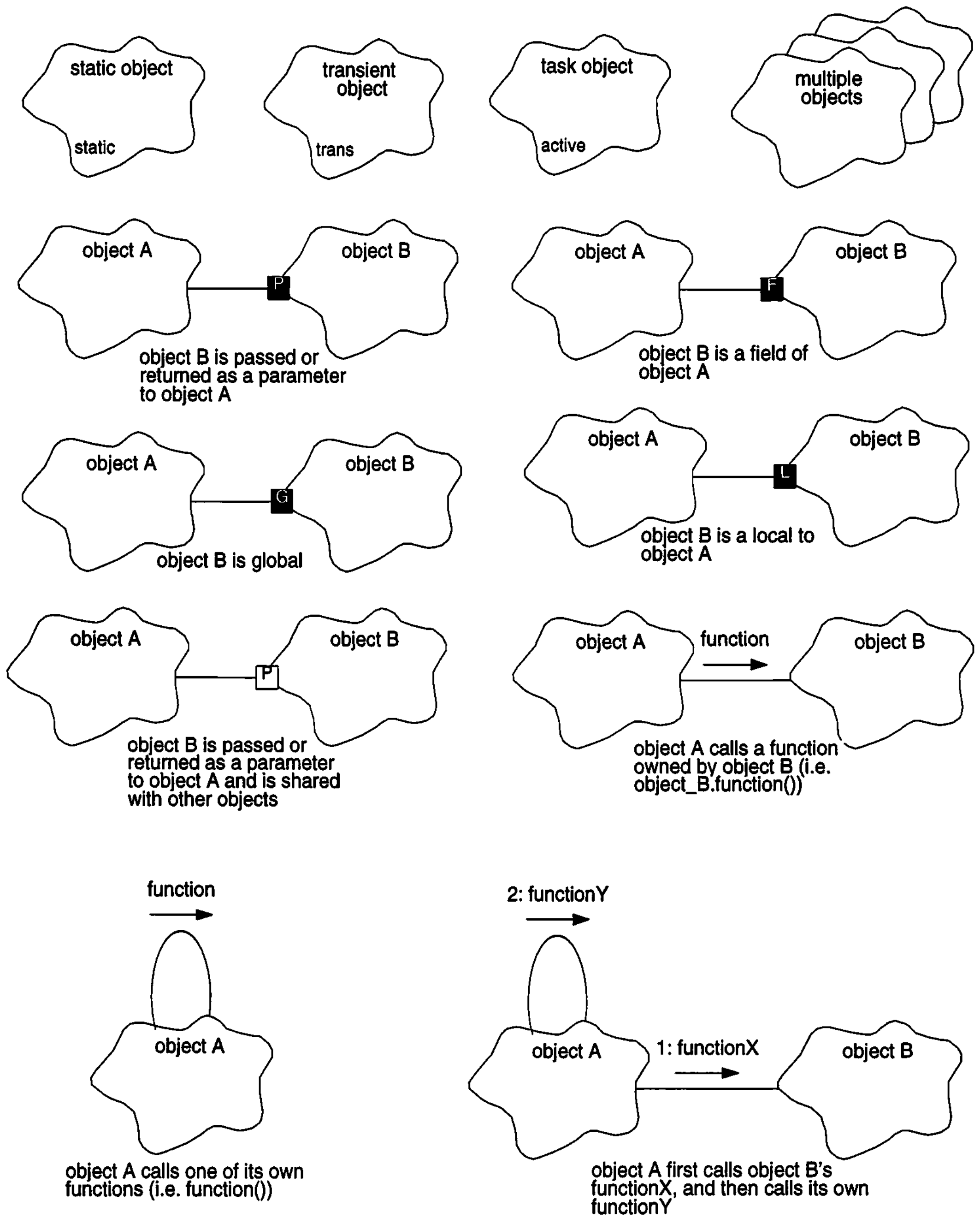

Class
public function( )
I protected function( )
II private function( )
Abstract Class
A
Class A
Class B
Class A is a subclass of Class B
Class A uses Class B as part of its public interface
Class A uses Class B in its implementation
1
Class A contains 1 public instance of Class B
Class A contains 1 public pointer to an instance of Class B
Class A contains a protected instance of Class B
Class A contains a private instance of Class B

Figure 2 illustrates the icons and relationships this document uses to illustrate scenarios involving different objects (i.e. class instances).

**FIGURE 2. Object Diagram Icons**


static object
static
transient object
trans
task object
active
multiple objects
object A
object B
P
object B is passed or returned as a parameter to object A
F
object B is a field of object A
G
object B is global
L
object B is a local to object A
P
object B is passed or returned as a parameter to object A and is shared with other objects
function
object A calls a function owned by object B (i.e. object_B.function())
function
object A calls one of its own functions (i.e. function())
2: functionY
1: functionX
object A first calls object B's functionX, and then calls its own functionY

## 2.6 Typographic and Naming Conventions

### 2.6.1 Class Category Names

All words and abbreviations of all class category names start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters.

All names of class categories (i.e. subsystem) within this document are represented using bold, italicized Courier text:

***`ClassCategoryName`***

### 2.6.2 Class Names

All words and abbreviations in all class, structure, enumeration, and union names start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters.

All class names within this document are represented using bold Courier text:

**`ClassName`**

### 2.6.3 Function Names

The first word or abbreviation in all function and member function names shall start with a lower-case letter. All remaining words, if any, within the name shall start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters. All function argument names shall conform to the convention described for Variable Names (see Section 2.6.4 ).

Functions which suspend execution of the currently running process until some condition is satisfied shall contain the word “wait” embedded in their name. Functions which require some condition, but do not suspend the process shall contain the word “request” embedded in their name. Functions which contain an infinite loop, such as the main function of a process, shall have the word “go” contained within their name.

All function names are represented using plain Courier text:

`functionName`

or

`functionName()`

### 2.6.4 Variable Names

The first word or abbreviation in all function and member function names Shall start with a lower-case letter. All remaining words, if any, within the name may or may not start with a capital letter. The remaining letters in each word of the name are in lower case. All words and abbreviations within the name shall contain at least two letters.

All variables, instance variables and structure or enumeration tags are represented using italicized Courier text:

*`variableName`*

### 2.6.5 Enumeration Tags

With the exception of the Boolean type, all letters in enumeration tag names are capitalized.

This document uses no special typographic convention for displaying enumeration tag values:

ENUMTAG (except for BoolTrue and BoolFalse, see Section 2.7 )

### 2.6.6 Preprocessor Definitions

All preprocessor definitions are completely capitalized. Spaces may be indicated using underscores.

This document uses no special typographical convention for displaying preprocessor definitions:

PREPROCESSOR_DEFINITION

### 2.6.7 Directory and File Names

Except for filename extensions, all directories and filenames are in lower-case, with spaces represented as underscores. All C-language source and header files use the ".c" and ".h" extensions, respectively. All C++-language source and header files use the ".C" and ".H" extensions respectively. All assembler files have the ".s" extension.

This document uses no special typographical convention for displaying directory or filenames:

directory_name/filename.C

## 2.7 Global Data Types

### 2.7.1 Bit and Byte Ordering

Given that there are no portability requirements on the ACIS software, unless otherwise specified, it assumes that all data elements have little-endian byte ordering. For example, the 32-bit value 0x12345678 is stored as bytes in RAM as follows:

| **Virtual Address** | **Byte Value** |
|---|---|
| 0 | 0x12 |
| 1 | 0x34 |
| 2 | 0x56 |
| 3 | 0x78 |

By convention, bits within a word are numbered with the least-significant bit as bit number 0. This is consistent with the "MIPS Programmers Reference Guide."

Unless otherwise specified, all signed values use two's complement representation.

### 2.7.2 Integer, Pointer and Enumeration Data Types

Given that there are no portability requirements on the ACIS software, rather than provide renamed type definitions, the ACIS software assumes that the compiler defines its concrete data types as follows:

**TABLE 2. Concrete Data Type Assumptions**

| **C/C++ Type(s)** | **Sign and Size** |
|---|---|
| char | signed 8-bit value |
| unsigned char | unsigned 8-bit value |
| short | signed 16-bit value |
| unsigned short | unsigned 16-bit value |
| int or long | signed 32-bit value |
| unsigned, unsigned int, or unsigned long | unsigned 32-bit value |
| all pointers | unsigned 32-bit values |
| enumerated types | unsigned 32-bit values |

### 2.7.3 Boolean Data Type

In order to distinguish true and false arguments and return values from signed integers (int), the ACIS instrument software defines a **Boolean** data type using the following enumeration:

```
enum Boolean
{
   BoolFalse = 0,
   BoolTrue = 1
};
```

This type is used by the software whenever an argument or return type expresses a true or false expression. Since, in C and C++, enumerated types can be converted to an integer type by the compiler, these enumerated values are compatible with compiler generated relational expressions. NOTE: In C++, relational expressions, however, can not be converted to a **Boolean** type without an explicit cast or conversion.

For example:

```
Boolean result = BoolTrue;
if (result == (3<4))      /* ok, result is compared as an int */
{
}
result = (3<4);           /* wrong, int not converted to Boolean */
result = (Boolean) (3<4); /* ok, explicit type cast */
```

## 2.8 Global Units

### 2.8.1 ACIS Timestamp Units

The ACIS timestamp counter relies on the spacecraft-supplied 1.024 MHz clock. As such, all timestamp values are in units of 0.9765625 microseconds.

### 2.8.2 Timer Tick Units

All timer tick references made within this document are in units of 100 milliseconds (the Back End Processor timer-tick period).

## 2.9 In-line Assembler

For convenience, some small, isolated portions of the instrument software use in-line assembler directives. These directives are compiler-dependent. The delivered version of the software will use the GNU C++/C Compiler, g++. This compiler allows for incorporating in-line assembler directives into its optimization strategies. As such, care must be taken to ensure that the instructions specified are executed in the order intended by the programmer.

The syntax for an in-line assembly directive for g++ is as follows:

```
asm [volatile] ("instruction(s)" : outputs : inputs[: registers]);
```

The "volatile" option tells the compiler that the instruction has side-effects not necessarily visible to the compiler, and tells it to avoid reordering or removing the instruction when optimizing. "instruction(s)" contains 1 or more assembly language instructions. Input and output variables in the instruction are delimited by %0, %1, %2, etc. The digit in the variable specifies the order of "outputs" and "inputs" options following the instruction. The "outputs" section contains a comma separated list of output variables constraints. The first constraint corresponds to the variable %0 in the instruction list, the second to %1, and so on. The constraints map a C or C++ language variable into the instruction. The "registers" option specifies a list of CPU registers clobbered by the instruction. Unfortunately, the current version of g++ does not support specifying co-processor registers in this list, and a "volatile" directive must be used if a system co-processor register is modified.

For example:

```
unsigned setStatusReg (unsigned value)
{
   unsigned retval; /* old status register value */

   asm ("mfc0 %0, $12" : "=d" (retval) : /* no inputs */ : $12);
   asm ('mtc0, %0, $12" : /* no outputs */ : "d" (value) : $12);

   return retval;
}
```

This above function does not work with optimization turned on. The "$12" register does not refer to an R3000 core register, but to one of the system coprocessor registers, and the compiler will not realize the dependence between the two instructions, and may reorder them. To avoid this, the "volatile" directive is used to tell the compiler that there are side-effects to the instructions that its optimizer does not know about. The correct implementation is as follows:

```
unsigned setStatusReg (unsigned value)
{
   unsigned retval; /* old status register value */

   asm volatile ("mfc0 %0, $12" : "=d" (retval) : /* no inputs */);
   asm volatile ('mtc0, %0, $12" : /* no outputs */ : "d" (value));

   return retval;
}
```

## 2.10 Nomenclature

### 2.10.1 Classes and Objects

In object-oriented design, there are "objects" and "classes." A class refers to the equivalent of a data structure definition and a collection of functions which operate on this data structure. An object refers to a physical declaration of a class. For example, in "C" one can have a structure such as:

```
struct foo
{
    int a;
};
```

`struct foo` is analogous to a class definition where as in declaration of one of these structures, such as in the case of:

```
struct foo bar
```

`bar` is analogous to an object declaration.

Within this document, the **Architecture** section describes the system primarily in terms of specific objects. Later portions of this document develop and describe the various classes within the Back End Processor software, and focus on the detailed behaviors of these classes.

### 2.10.2 Class Instance Lifetimes

The class descriptions in this document specify the lifetime of an instance of the class using two keywords:

- *Persistent* - Instances of this class exist for the lifetime of the program (i.e from startup to shutdown)
- *Transient* - Instances of this class may be created and destroyed as the program runs.

The use of the word *persistent* is a little misleading, in that it implies that the state of an object may be retained across instrument resets. Although there are a few such items, this document uses the keyword to indicate objects which are created at startup, and which last until the instrument is reset. All instances of a *persistent* class must last for the duration of the program. *Transient* objects are those which are created and initialized as needed by the running program, and destroyed when they are no longer in use. Some objects defined from *transient* classes may exist for the lifetime of the program, and others may come and go as the program runs. Unlike most C++ programs, the ACIS instrument software does not use a global run-time memory heap in order to avoid heap fragmentation and other issues. The memory for *transient* objects is obtained from collections of memory buffers, or on the stack. The lifetime of an object allocated from a memory pool ends when the object's destructor is invoked, and its memory is released back to its pool. The lifetime of an object declared on the stack ends when the scope of the code block which declared it ends.

### 2.10.3 Class and Function Concurrency

The class and function descriptions in this document specify the type of task and interrupt environment supported by the class or function, using three keywords:

- *Sequential* - Correct operation is guaranteed if only one task has access to the class or function
- *Guarded* - Correct operation is guaranteed if multiple tasks coordinate access to the class or function
- *Synchronous* - The class or operation performs all operations needed to coordinate access by multiple tasks. No special operations are required by the client.

In general, classes and functions used only during initialization, assume that only one thread of control is active, and specify *Sequential.* Classes and functions which expect to be used by only one task, or expect multiple tasks to arbitrate amongst themselves for access to the class or function, specify *Guarded.* Classes and functions which either do not require any inter-task arbitration (i.e. do not directly or indirectly modify any shared variable or hardware), or perform the arbitration internally, specify *Synchronous.*