A Successful Example of a Layered-Architecture Based Embedded Development with Ada 83 for Standard-Missile Control

Kelly L. Spicer
Raytheon Missile Systems
Missile Software Engineering Center
Tucson 520-663-7020
klspicer@west.raytheon.com

1. Executive Summary

A need for a standard understandable software architecture that can be reused from project to project has long been recognized. This paper describes the software architecture used for the Standard-Missile-3 (SM-3), Stage-3 development program. The architecture was defined based on hierarchical principles with the goal of providing a general solution for the architecture-level design for embedded systems. An architecture-need statement is first presented listing the attributes of the needed architecture. An overview of the notation is given, and then the layered architecture is described. How the architecture supports multi-build projects is then described. A brief discussion on reuse is then provided – both reuse of algorithms in the architecture and reuse of the architecture itself. The paper concludes with a few comments on support for the software process. The paper includes two attachments: Attachment A - a list of acronyms and other definitions, and Attachment B - an overview of the SM-3 Stage-3 software requirements.

2. Introduction

Embedded software often is almost completely custom software. These systems are too often built from scratch or informally pieced together from parts of other similar applications that a programmer or software manager may be aware of.

A domain-specific software architecture is needed to bring more engineering discipline and predictability to software development. Product cycle times and the costs of new development could be reduced through its use; software quality and reusability would be increased.

3. Software Architecture Need Statement

The desired architecture would exhibit the following attributes:

a. Quickly adaptable to new projects within the application domain.

b. Simple to grasp by both managers and developers.
   - Managers need to be briefed on the design so they understand how it will implement the requirements.

c. Includes specific places in the architecture to map requirements:
   - mission-functional requirements.
   - project-performance requirements.
   - project-hardware requirements.

d. Includes a clear place for defining tasks and doing systems-type performance tuning.
   - task priorities, task stack allocations, etc.

e. Isolates hardware and external interface dependencies (IRS material) from mission algorithms (SRS material).

f. Includes a simple, easy to understand notation.

g. Includes conventions for maintaining the integrity of the architecture during development.
   - Rules that can be checked in design and code reviews.

h. Easily lends itself to multi-build development projects.

4. Layered Architecture Overview

The SM-3 Stage-3 Software is implemented in the Ada programming language. Ada constructs are well suited for implementing the principles and rules of the architecture, but not essential for its use.

4.1. Architecture Design Notation

The design notation used for the architecture and the subsystem designs is a modified Buhr. Figure 1 depicts the notation used.

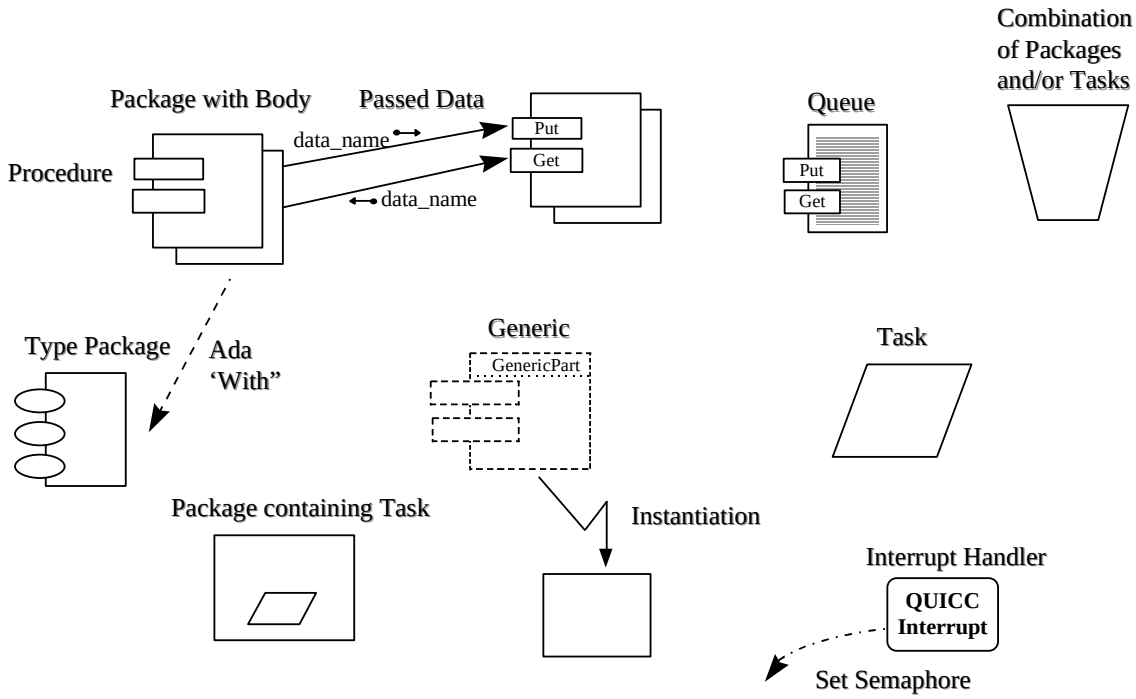

Figure 1 - Modified Buhr Notation

4.2. Layer Descriptions

This software architecture utilizes a layering scheme. The scheme characterizes a Seniority-Hierarchical-Layered approach. In this approach each layer is, in essence, a virtual machine for the layer(s) above in the hierarchy. The layering principles are similar to those used in the ISO/OSI Reference Model. The following principles and rules are among those used in defining the architecture:

a. Hierarchical approach
b. Lower layers provide services and implement interfaces
c. Upper layers implement mission algorithms and maintain mission state
d. Commands flow down
e. Data flow up and down
f. No direct software dependencies "up' the architecture; down only.
   - The lower layers should have no dependencies on higher layers. This is key to the reuse of the architecture.

There are six layers in the architecture. The layers are named and described, from the top layer down, as follows (see also figure 2):

1. **Timing Control** - This layer is used only to control the rates for the Mission Tasks. These tasks are periodic. To meet throughput requirements the beginning of each frame is synchronized with the arrival of periodic data.

2. **Mission Task** - This layer implements the task structure and task-to-task sequencing for the Mission-Support layer.

3. **Mission Support** - This layer implements most or all of the mission algorithms defined in the SRS. These algorithms call down to the Mission-Interface layer to send messages, fire squibs, etc. The software packages are grouped into Functional Subsystems (i.e. autopilot, guidance, mission sequencing; Functional Subsystems are discussed in a later section). These are called to "update" periodically by the Mission-Task layer above. Direct dependencies between the Functional Subsystems are avoided; data are passed from one to the other via a common Store-Manager package. The Store-Manager package ensures data integrity by preventing simultaneous access.

4. **Startup** - This layer drives processing required only at startup time (starting interfaces, conducting BIT, etc.). The tasks in this layer terminate after startup time, so this layer logically goes away then.

5. **Mission Interface** - This layer provides a logical and consistent interface between the Mission-Support and Startup layers above, to the hardware interfaces and message processing below. The package specifications for each interface are built from a common template to ensure consistency. This layer, along with the Hardware-Interface layer below, implements the interface protocols defined in the IRSs. This layer understands the external message formats (defined in the IRSs). Received messages are parsed and converted to the internal format and the data

are placed into the Store-Manager package for use by mission algorithms in the Mission-Support layer. Generics are often used for message handling.

6. **Hardware Interface** - This layer implements the actual hardware interfaces. It doesn't understand internal message formats; instead, outgoing messages are converted to strings of words by the Mission-Interface layer above and passed down to be transmitted over the interface. It does understand hardware addresses, register formats, low-level hardware timing requirements, interrupt handling, etc. Generics are used to handle more than one interface when they are sufficiently similar.

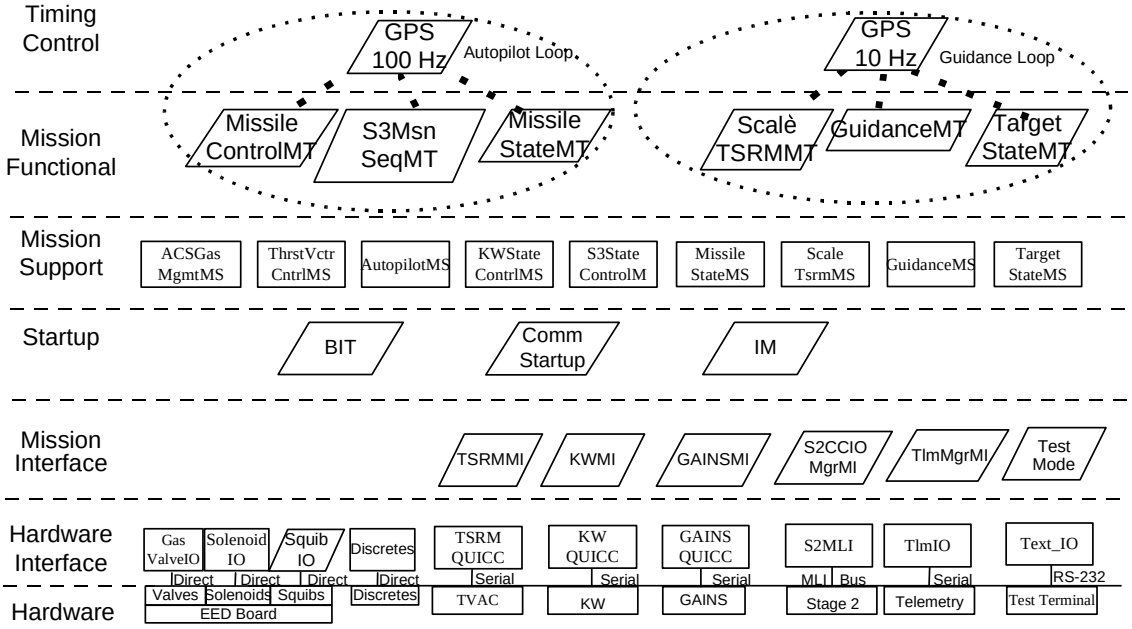

Figure 2 - Layered Architecture Overview

Internal and external data formats are defined as follows. Each functional subsystem includes an interface-definitions package that defines the internal data passed between subsystems. These data are passed via the gets and puts to the Store Manager. For external messages, each interface provided by the Mission-Interface level includes a message-definitions package that defines the formats of these messages (these formats map directly from the IRSs).

4.3. Architecture Conventions

4.3.1. Naming Conventions

Naming conventions associate a component's position in the hierarchy and functional position. For example suffixes and nmemonics are assigned the following way:

- Mission-Task Layer : "MT" Suffix
- Mission-Support Layer : "MS" Suffix
- Mission-Interface Layer : "MI" Suffix
- Interface-Definitions package : "IntfcDefs" Suffix
- Subsystem mnemonic : (e.g. "ThrstVctrCntrl" for Thrust-Vector Control)

Package names are formed by concatenating the subsystem mnemonic to the appropriate suffix. e.g.:

-ThrstVctrCntrlMS
-ThrstVctrCntrlIntfcDefs

4.3.2. **Layer-Interface Conventions**

Layer-Interface conventions ensure that a layer's components export a consistent interface to the layers above. For example:

Each Mission-Interface package exports these procedures:

-"procedure Startup (Startup_Complete : out boolean);"

Called by the CommStartup Task until true is returned indicating the interface startup protocol defined in the IRS has completed.

-"procedure Send<MessageName>;"

One for each message able to be sent on the Interface. Parameterless, data retrieved from Store Manager.

Each Hardware-Interface layer package exports at least the following:

-"procedure Initialize (Startup_Complete: out boolean);"

Initializes hardware. Generally called by the Startup procedure at the MI level as part of its processing.

- Functional Procedures to conduct Hardware Operations
  (i.e. “FireSquib,” “TurnValveOn,” etc.)

Similarly for the Mission-Support layer, each package exports an initialize procedure and an update procedure. Each “Initialize” is called once at startup time by the Mission-Task layer, and each “Update” is called once per periodic frame.

4.3.3. Data Flow up the architecture

Because there are no dependencies (Ada “withing” or procedure calls) up the hierarchy, you may ask: “how are incoming data and events passed to the higher-level packages?” The answer is semaphore signals, queues, and the Store Manager. For example in Figure 3, when a message arrives from Stage-2, the interrupt handler in S2MLI copies the data from the hardware to a queue and signals the task in S2CCIOMgr, this task then “wakes up” (at its priority), dequeues the data, converts in from external to internal format, and places the data in the Store Manager for use by other Subsystems.

4.4. Functional Subsystems

Packages are logically grouped into functional subsystems. An example of one of the Mission-Interface layer subsystems (Stage-2 Interface), and its context in the layered architecture is depicted in Figure 3 (stage-2 Interface packages are darkened).

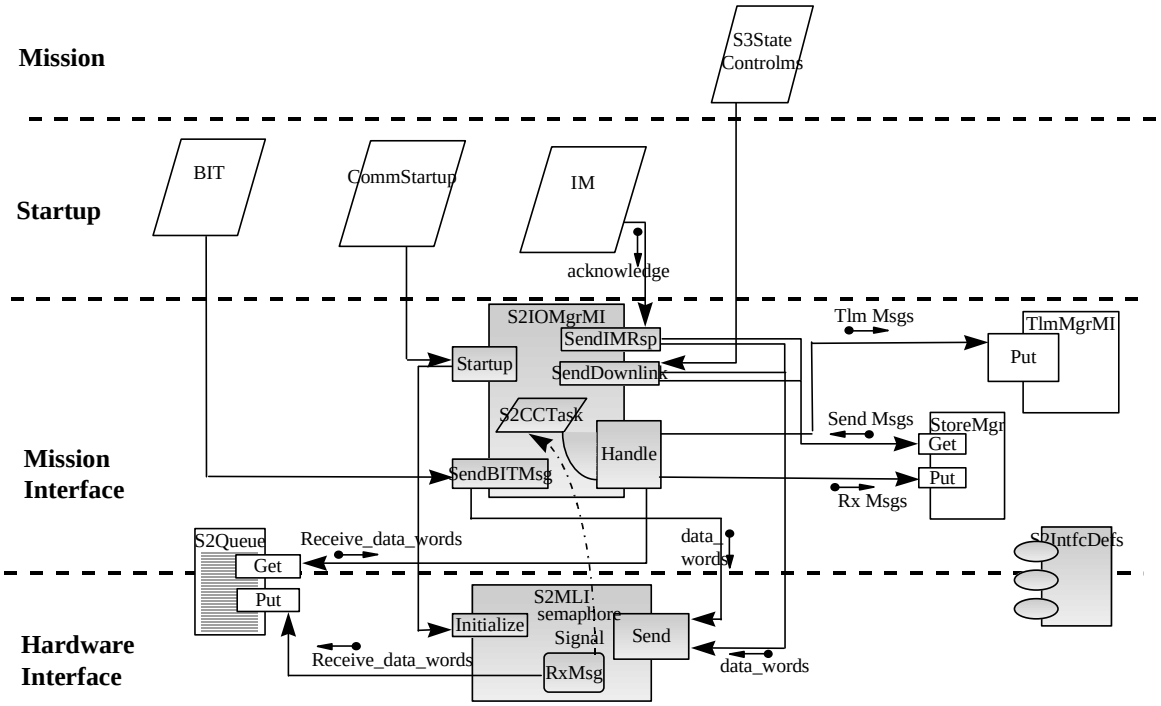

Figure 3 - Stage-2 Interface Functional Subsystem

Figure 4 depicts an example of a Mission-layer subsystem, State Control, and its context in the layering scheme.

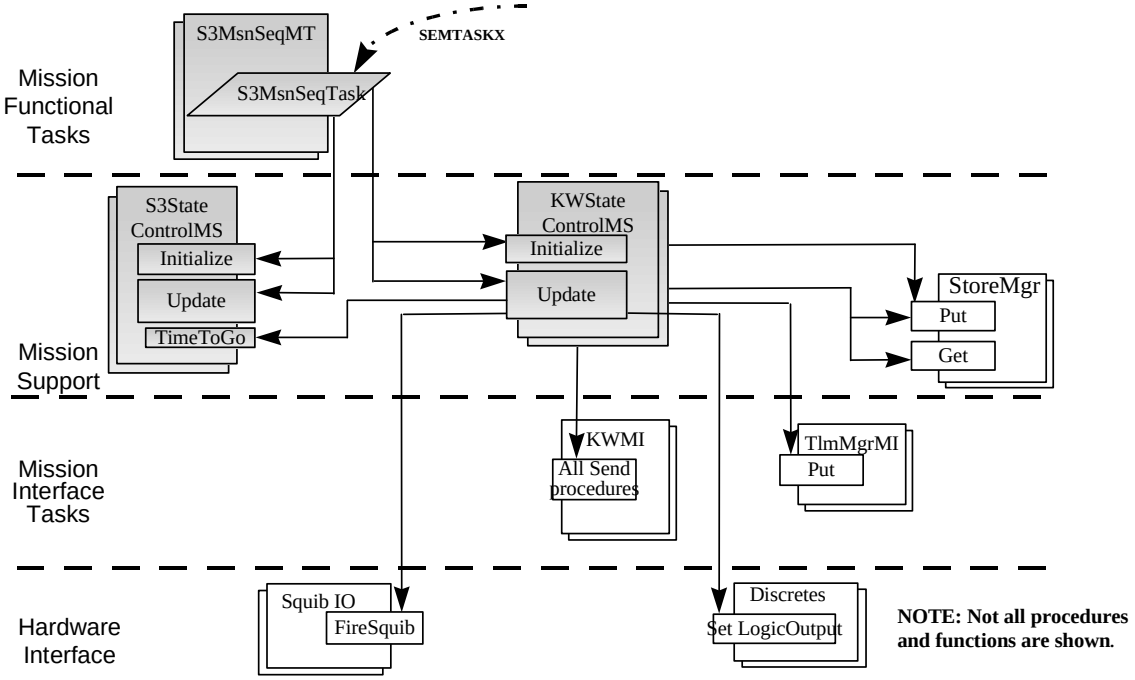

Figure 4 - Stage Control Functional Subsystem

5. Using the Architecture on Multi-Build Projects

An example of using this architecture on a multi-build project would be one where the interface layers (lower layers) are built first; and the Mission layers (upper layers) are built later. In this example, the order for completing builds to construct the system would go as follows:

1. First build the interface layers, implementing IRS Requirements (lower layers).
2. Build the "Architecture Skeleton" for the rest of the system (upper layers).
3. Add the implementation/mission details incrementally (from SRS).

Figure 5 depicts the first of these three builds, as was done on SM-3 Stage-3 project. The darkened icons were implemented in the first build. The open boxes were left for later builds. The Build-1 Sequencer is the test driver used to test all the interfaces.

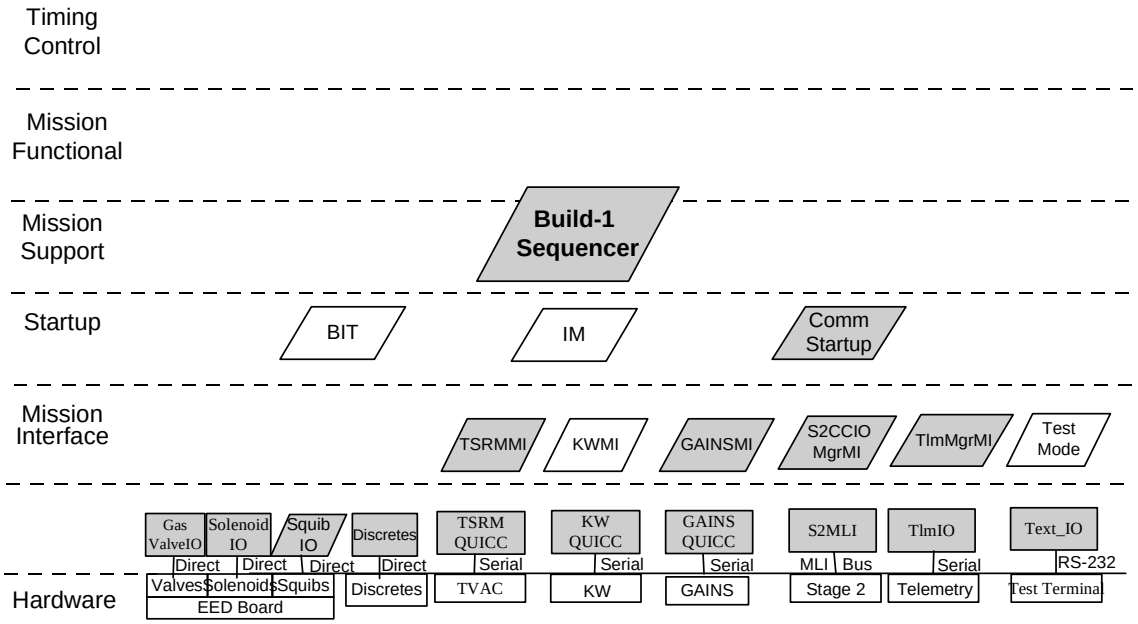

Figure 5 - First Build in a Multi-Build Project

This approach also provided an early build that was used by hardware-checkout personnel to help validate the hardware requirements.

After Build 1 verified the interface layers against IRS requirements, the first follow-on build was conducted to complete the remainder of the "Architecture Skeleton." This included implementing all the packages and tasks but without the mission algorithms.

Intertask interfaces were established, task priorities, etc. were all implemented in this build. The full architecture could then be executed on the target with stubs for the algorithms.

The purpose was to provide a stable architecture baseline. SRS-defined algorithms could then be implemented in later builds without concerns for architecture-level software design, inter-task communication, or external interfaces. Also, since the Mission-Support packages have no direct dependencies on each other (remember they don't "with" each other since data are passed via a Store Manager), the order of the algorithm development could be planned based on test schedules. For example, the autopilot algorithms could be implemented first and used to test the stability of the airframe. Guidance algorithms could be implemented later.

6. Reuse of Algorithms

By decoupling the algorithms from the external interfaces and other algorithms in the Mission-Support layer, they could be reused between other projects. For example on the SM-3 Stage-3 project many of the algorithms were reused from a predecessor project, Terrier LEAP.

7. Reuse of the Architecture

It is suggested that this architecture represents an Architype for Embedded Missile System (EBMS) software development. The following goals exist for reuse of this architecture:

1. The application could be ported to new hardware by changing only the Hardware-Interface layer and parts of the Mission-Interface layer of the architecture.
2. The architecture could be quickly reused on other EBMS applications in the same manner it was used on SM-3 Stage-3 LEAP as described in this paper.
3. The concepts of this layered-architecture approach would be useful in defining other domain-specific software architectures, especially for embedded control applications.

8. Project Status

The extensive ground test program is completed and the flight test program is underway. The third flight is scheduled for Dec 2000. No significant Stage-3 Software problems

have been found. During development no significant design issues were raised. No changes were needed to the initial architecture-level design diagrams.

9. Conclusion

This paper presents a layered software architecture used to implement the SM-3 Stage-3 Software for the AEGIS LEAP Intercept program. The paper proposes that this architecture, or design principles upon which it is based, satisfies a more general need for a standard software architecture for embedded control systems. Details of the architecture are described and its use on a multi-build project are also described. Use of this architecture has proven successful on the SM-3 program.

Kelly Spicer

Kelly was the Software Development Lead, and later, Software Team Lead for the Standard-Missile-3/Stage-3 Program during software development and testing. He has been with Raytheon Missile Systems Company in Tucson for five years.

Previously, as an Air Force Officer, Kelly was a software designer and team lead for other programs, more recently with Air Force Space Command supporting NORAD in Colorado Springs. Among his positions was the Lead for "Domain-Specific Software Reuse" on the Air Force/STARS Technology Demonstration Project (SCAI). This technology-insertion project was seeking to utilize new process, reuse, and software-engineering-environment technology, developed under DARPA contracts, on an actual Air Force program.

Kelly received his masters degree from the Air Force Institute of Technology. His thesis focused on defining software reuse architectures for embedded systems (Available from the National Technical Information Service 1-800-553-6847; the order number is AD-A230 460/8INZ.). He also published a paper in the ACM's "Ada Letters" on the same subject (Nov-Dec 1991 issue).

Attachment A
Acronyms & Definitions

ACM	Association for Computing Machinery
AEGIS	Advanced Electronic Guided Intercept System
BIT	Built-in-Test (or Built-In-Test Handler)
CMM	Software Engineering Institute's Capability Maturity Model
DARPA	Defense Advanced Research Projects Agency
EED	Electro-Explosive Devices
GAINS	GPS-Aided Inertial Navigation System
GPS	Global Positioning System
IM	Initialization Message (or Initialization message handler)
IMU	Inertial Measurement Unit
ISO/OSI Reference Model	Internal Standards Organization/Open Systems Interconnect Layered Model
KW	Kinetic Warhead
MLI Bus	Multi-Level Interface Bus (used as the hardware interface between stage 3 and stage 2)
NORAD	North American Aerospace Defense Command
QUICC	Motorola's MC68360 Quad Integrated Communications Controller
SCAI	Space Command & Control Architectural Infrastructure
SDACS	Solid Divert Attitude Control System
Semaphore	A service provided by the software kernel. Semaphores are generally used to signal a task to run. The code "waits" at a semaphore name until another task "sets" that semaphore name.
SM-3	Standard Missile-3 Program, also known as AEGIS LEAP Intercept Program
Stage-2 Control Computer	Controls the missile from booster burnout to Stage-2 burnout
Stage-3 Control Computer	Controls the missile from Stage-2 burnout to eject of Kinetic Warhead
STARS	Software Technology for Adaptable Reliable Systems (DARPA funded program)
TSRM	Third-Stage Rocket Motor
TVAC	Thrust Vector Activation/Control

Attachment B

Overview of the AEGIS LEAP Intercept Software Requirements

I. The AEGIS LEAP Intercept Program

The principle objective of the AEGIS LEAP Intercept (ALI) (also known as Standard-Missile-3 or SM-3) is to develop and demonstrate a capability for engaging and intercepting tactical ballistic missile (TBM) threats in the exoatmosphere from an AEGIS launch platform. This program is part of the overall Navy Theater Wide (NTW) development effort to provide the fleet with capability to defeat medium to long-range theater ballistic missiles.

II. The SM-3 Stage-3 Software Requirements

The application example for this paper is the Stage-3 software for SM-3 missile. Stage-3 is one of five CSCIs developed for this missile. Each CSCI is developed by a different software team. These CSCIs and their interfaces are depicted in figure A-1.

The following is a brief summary of the Stage-3 software requirements:

1. Interfaces
   - Serial (Third-Stage Rocket Motor (TSRM), GPS/IMU Navigation, KW, Telemetry)
   - Direct control (Attitude Control System (ACS), squibs, solenoids, descretes)
   - MLI bus (stage 2)
2. Mission Sequencing control

   Two Mission Squencer timelines are maintained:
   - Transitions through mission events based on time-since launch
   - Transitions through events to prepare the Kinetic Warhead for ejection based on time to go.
3. BIT - functions and reporting
4. Autopilot/attitude control

   There are three modes of control:
   - Cold gas

- Warm gas
- Thrust Vector Control (TVC)

5. Guidance
   - Burnout reference guidance
   - Adjust for non-nominal rocket-motor burn-rate

6. Adaptation parameters
   - Capability to load key constants separately from the software

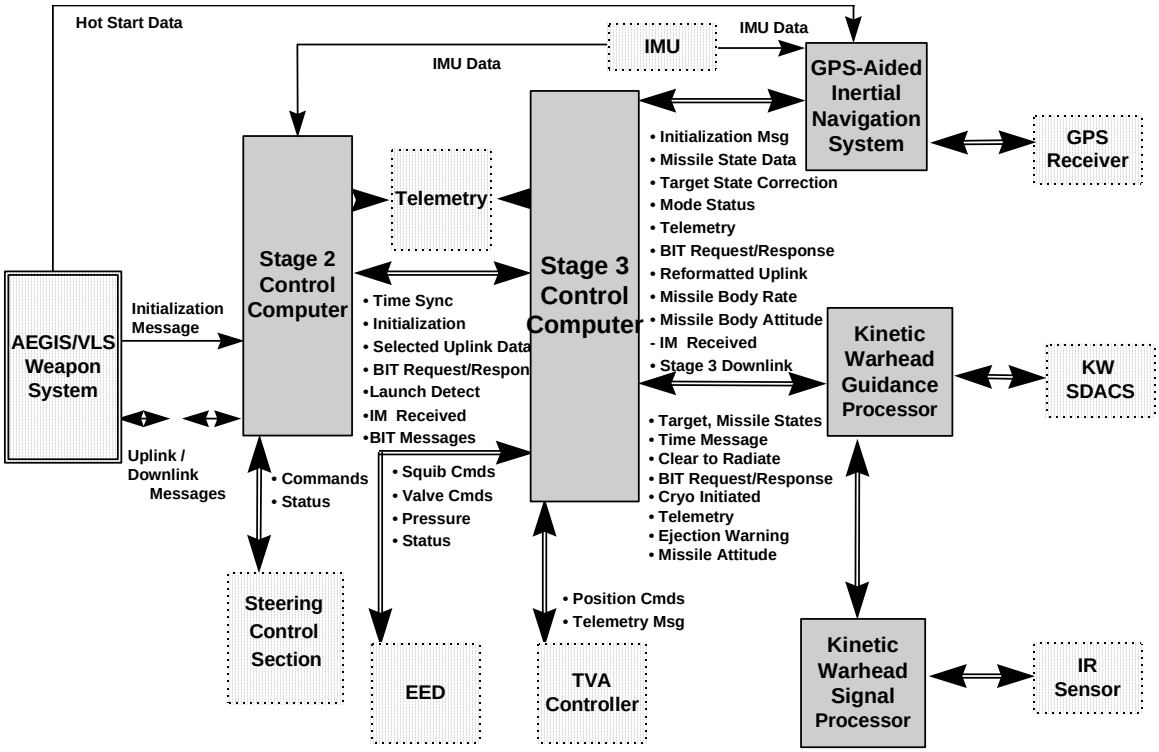

Figure A-1 - SM-3 CSCIs

Note: See attached definitions