KAPL, Inc.
Knolls Atomic Power Laboratory
Post Office Box 1072 Schenectady, N.Y. 12301-1072
Telephone (518) 395-4000 Facsimile (518) 395-4422

LOCKHEED MARTIN

SPP-67610-0007
July 30, 2005

The Manager
Schenectady Naval Reactors Office
United States Department of Energy
Schenectady, New York

Subject: Prometheus Reactor I&C Software Development Methodology, for Action

References:
(a) JPL Document: Prometheus Project – Project Software Management Plan (preliminary), 982-00046, Rev. 0
(b) Bettis Letter: Reformatted Software Engineering Policy, for NR Information, B-REO(M)-CD-008, 3/17/05
(c) KAPL Letter: All Projects: NNPP Equipment: Software Qualification by Criticality Level – Three-Prime Task Force Recommendation; For NR Approval, ARP-68640-0196, 4/12/02
(d) NAVSEA Letter: All Projects – Shipboard Software Qualification by Criticality Level – Three-Prime Task Force Recommendation; Approval with Comment and Request for Prime Contractor Action, Ser. 08K/03-00484, 2/6/03
(e) KAPL Letter: All Projects: NNPP Standard for Software Qualification by Criticality Level; For Concurrence, ARP-68640-0305, 9/2/04
(f) KAPL Letter: KAPL Comments and Concurrence to Proposed NNPP Standard for Software Criticality Level, FSO-64K20-04-143, 12/21/04
(g) BPMI Letter: All Plants – NNPP Standard for Software Qualification by Criticality Level; BPMI Concurrence, BPMI-ICS-PMP-00731, 3/11/05

Enclosures:
(1) Space Electrical Systems Software Life Cycle, Methodology, and Language Choice for Prometheus Reactor I&C Software Development
(2) NRPCT Reactor I&C Software Development Process Manual (NRPCT-RIC-SDPM-001)
(3) NRPCT Reactor Module Software Development Plan (NRPCT-RM-SDP-001)

Dear Sir:

Purpose:
The purpose of this letter is to submit the Reactor Instrumentation and Control (I&C) software life cycle, development methodology, and programming language selections and rationale for project Prometheus to NR for approval. This letter also provides the draft Reactor I&C Software Development Process Manual and Reactor Module Software Development Plan to NR for information.

PRE-DECISIONAL – For Planning and Discussion Purposes Only

Background:
As part of project Prometheus, the NRPCT has been working with other team members (JPL, NGST, and Hamilton Sundstrand) to create a set of high level process requirements and principles for software development that would allow for better communication and commonality between the various software efforts within the Prometheus program. These process requirements have been gathered in preliminary form in the Reference (a), Project Software Management Plan (PSMP). The guidance provided in the PSMP would then be expanded for each team organization in a local Software Development Plan (SDP), which would trace back to the PSMP and any local organizational requirements.

As part of developing the Reactor I&C SDP, different software life cycles were examined to help define the software development process. Different design methodologies and languages were also examined for appropriateness. These comparisons lead to the selections provided in Enclosure (1) and helped define the processes and development plan in Enclosures (2) and (3).

As part of the process for developing software for the Naval Nuclear Propulsion Program (NNPP), the software Criticality Level (CL) must be defined for each deliverable. The qualification of software by CL was originally proposed as part of a three-prime task force recommendation to NR in Reference (c). NR approved the task force recommendations with comment in Reference (d). The NR comments were incorporated into the NNPP Software Qualification by Criticality Level (SQCL) document and distributed for three-prime concurrence by Reference (e). KAPL and BPMI concurrence to the SQCL is documented in References (f) and (g). Bettis concurrence to the SQCL is pending.

Discussion:

Enclosure (1) provides the options and rationale for the selection of a software life cycle, design methodology, and programming language for NR approval. The selections consist of the Incremental software life cycle, structured design methodology, and C programming language for Prometheus reactor I&C software development. After a comparison of the major software life cycles that have been defined for use with different software development activities, the NRPCT has selected the Incremental life cycle for use with Prometheus Reactor I&C software development. The Incremental life cycle provides for a series of software releases that provide increasing functionality to afford earlier opportunities for integration with other software components. This will allow performance of interface testing and help mitigate risk. The selection of a Structured design methodology allows for a robust software architecture design making use of top-down design, functional decomposition (hierarchical refinement of functionality from a course level of detail to a fine level of detail), and structured programming. This allows for strong modularity in the design while avoiding some of the inspection burden associated with object-oriented design methodologies. The selection of the C programming language complements the use of structured design. Additionally, the C language has been used in many space applications and minimizes the inspection burden that may be associated with languages such as C++.

Enclosure (2) provides the draft software development processes based on the Incremental software lifecycle. These processes incorporate guidance from both the Reference (a) PSMP, and the Reference (b) Naval Reactors Software Engineering Policy. The Incremental lifecycle is defined in a series of tasks, starting with initial requirements and architecture development, through increment planning. The development tasks applied within each increment include:

detailed requirements development, detailed architecture development, module design, module implementation, unit testing, integration, system test, release, and independent verification and validation. Processes have been identified for each of the development tasks, and for wide-ranging tasks such as configuration management and defect tracking.

Enclosure (3) provides the draft Prometheus Reactor Module Software Development Plan (SDP) and is a subordinate document to the NRPCT Software Development Process Manual (SDPM). The SDP provides mission specific definitions of project roles, deliverables, a documentation hierarchy, organizational division of responsibilities, and a description of tools. The draft SDP is a higher level SDP that would then have deliverable specific sub-tier development plans tracing up to it. It is envisioned that the flight software, ground software, and test beds would each have a development plan that would trace up to the Reactor Module SDP. Below each subordinate SDP would also be a work breakdown structure and schedule specific to each deliverable. These subordinate deliverable specific SDPs will be developed and recommended in future submittals.

It will also be necessary to define the software Criticality Level for each NRPCT deliverable. As described in the Background, the SQCL is currently out for three-prime concurrence. Once the mission has been fully defined for Prometheus, the Criticality Level for each NRPCT deliverable will be assigned and justified. This will be provided to NR for approval via separate correspondence.

Extensibility to Lunar Mission:

Enclosures (1) and (2) were developed to be mission independent and as such, the major conclusions reached concerning the software development life cycle, software design methodology, and the selection of a programming language are fully extensible to a lunar space reactor system. Enclosure (3) was developed specifically for the Prometheus deep space mission. As such, the detailed implementation described in Enclosure (3) is specific to a JIMO type mission, for a lunar space reactor system a separate software development plan containing the same format and kind of information would be developed.

Conclusion:

NR approval of Enclosure (1) for the selection of the Incremental software life cycle, structured design methodology, and C programming language for NRPCT Prometheus Reactor I&C software development is requested.

Enclosure (2) and Enclosure (3) for the draft NRPCT Software Development Process Manual and Software Development Plan are provided to NR for information.

Definition and justification for the NRPCT Reactor I&C software Criticality Level, as documented in Reference (e), will be provided by separate correspondence.

This letter has been reviewed and concurred to by the Manager of KAPL SPP Space Electrical Systems – Systems and Software Design (M. Ryan), and the Manager of Bettis Space Instrumentation & Control Design (D. Robare).

Very truly yours,

Thomas A. Hamilton

Thomas A. Hamilton, Engineer
Space Electrical Systems – Systems and Software Design
Space Power Program

Space Electrical Systems
Software Life Cycle, Methodology, and Language Choice for Prometheus Reactor I&C Software Development

Thomas A. Hamilton
David Schroeder
Brian Robinson

July 2005

This page intentionally left blank.

Table of Contents

1 INTRODUCTION

Software development in the NRPCT for the Prometheus project requires a high level of quality in both software and documentation. Design, integration, and V&V (verification & validation), processes and standards are defined and performed in order to provide this high level of quality. Additionally, documentation of the rationale for the choices in the software life cycle, methodology, and development language is necessary to provide the basis for these decisions, and to help train new developers that may enter the project. The software life cycle provides the framework and sequence for the requirements, design, implementation, and testing activities performed as part of software development. The methodology (object-oriented or structured) provides the approach to requirements development and software implementation. Programming language choice is influenced by many factors, including the chosen method and developer experience.

The NRPCT has chosen the Incremental software life cycle for the development of the Reactor Module Flight and Ground software. The structured design methodology has been chosen for requirements and software implementation, and the C programming language has been chosen for software implementation. The options considered and justification for each of these choices is presented in the following sections.

1.1 Definitions[1]

Design Methodology – One of several techniques used to approach software design, object-oriented design centers around coupling data with algorithms, structured design centers around top-down design with modularity (function call/return).

Process Model - A model of the processes performed by a system; for example, a model that represents the software development process as a sequence of phases. Process models have a focus on management and support activities (program management, configuration management, quality assurance, process definition, etc.)

Software Life Cycle – The period of time from the inception of a software project to the retirement of that software, including requirements, design, implementation, testing, deployment, and maintenance. The life cycle focuses on the technical activities needed to analyze, design, and implement the desired system.

1.2 Acronyms

Acronym	Definition
IEEE	Institute of Electrical and Electronic Engineers
JPL	NASA Jet Propulsion Laboratory
MC/DC	Modified Condition/Decision Coverage
NRPCT	Naval Reactors Prime Contractor Team
UP	Unified Process (Rational Unified Process)

[1] Definitions taken, in part, from IEEE Std-610

V&V	Verification and Validation

2 METHODOLOGIES

2.1 Software Life Cycle

Software development for project Prometheus is performed using a phased development process with incremental builds. These incremental builds provide increasing levels of functionality until the full functionality is provided in the final build. The Prometheus incremental build plan consists of software from all modules (Spacecraft, Mission, Reactor, and Ground). This creates the desire for the Reactor Module I&C software to provide incremental builds to fit with the Prometheus overall incremental build plan as developed in discussions with various other Prometheus software development organizations. This implies that communication functionality should be present in early increments to be able to interface with the other modules.

Although there is a need to provide software in various increments to the various Prometheus team members, there is still a large degree of latitude for the NRPCT to choose the optimal software life cycle to develop the Reactor Module software. Several life cycles have been analyzed including the Waterfall, Spiral, Evolutionary, Rational Unified Process (UP), and others. Each of these life cycles are presented briefly, with the criteria weighted and a final choice presented.

2.1.1 Waterfall Life Cycle

The waterfall life cycle is often considered the "traditional" model for software development. The waterfall consists of a systematic flow from step to step. The various steps are as follows:

1) Requirements elicitation and development – The functional requirements for the system are developed. This includes gathering user requirements, analyzing and developing the system requirements from the user requirements, and documenting these requirements.
2) Software design – Once the requirements have been established, software design can be performed to lay out the system architecture, define all of the software modules, and assign functionality to each module. Either object-oriented or structured methodologies may be used. The design is documented in an appropriate format (UML, Flow Charts, etc.)
3) Software implementation - Once the design has been established and the modules are defined, these modules are then coded.
4) Integration - All of the discrete software modules are combined together to create the overall system. Integration testing is performed to ensure the validity of the combined modules.
5) Test – Testing is performed on the software system. Testing is performed in several phases (Unit Test, System Test, and Acceptance Test).
6) Operation – The system is released for use.
7) Maintenance – Any defects encountered are fixed. Functionality may be refined.

The waterfall model is one of the simplest to understand, but can be unrealistic since errors encountered may force rework of earlier tasks, and often requirements are in a state of flux and change late in the course of a project. The waterfall life cycle is most strongly applicable in an environment where the requirements are well defined early in the process and do not change. If requirements are in flux, this causes a great deal of iteration and rework which the waterfall model is not well equipped to handle. The documentation requirements for the waterfall method can be significant resource investments.

2.1.2 Spiral Life Cycle

The spiral software development life cycle is so named due to a cyclic series of development steps performed with increasing definition to the requirements each loop around the spiral. One of the main distinguishing features of spiral development is the institutionalized inclusion of risk analysis and risk management. Prototyping is also used to help define the requirements and mitigate risk.

Each transit around the spiral touches on four major areas:

1) Determine objectives, alternatives, and constraints,
2) Risk analysis and prototyping,
3) Requirements development and design,
4) Planning for the next spiral.

The risk analysis is used to ensure that risk can be managed or that if at any point during the development cycle, if the cost of risk mitigation is too great, the project can be cancelled prior to a full commitment of resources. The final loop around the spiral is very similar to the traditional waterfall. The prototype is discarded and a formal design is developed for the software product, flowing into an integration and test program.

Spiral development is most useful for high risk developments to allow a full vetting of risks prior to fully committing to design.

2.1.3 Incremental Life Cycle

In the incremental life cycle, the development work is focused on the construction of one part (subsystem) of the final product at a time. Each subsystem is finalized and planned for release in a specific version of the product. Each release is a functional version of the application containing more features of the final desired product than the previous increment. Within each increment, a waterfall development process may be followed to provide the requirements, design, and testing for that particular increment.

The project scope must be fully defined from the start. This allows the content of each release to be based on priorities set by the project team. The order of feature construction can be selected in different ways. For example, riskier functions may be chosen for earlier increments. The fundamental system architecture and some important requirements should also be implemented in earlier increments. The Incremental life cycle provides the ability to integrate early and often. This helps to minimize the impact of defects by finding them early in the life cycle.

2.1.4 Evolutionary Life Cycle

The evolutionary model defines and develops one piece of a system at a time. This allows the process to respond to change in the system requirements and add functionality as new requirements come to light. Typically this is accomplished through many short iterative development cycles. The basic functionality is developed first with additional features and functionality being implemented or modified as the requirements evolves. The evolutionary concept is incorporated in many of the other life cycle models.

While the evolutionary model is well adapted to handling changing requirements (since it is only focused on one portion of functionality at a time), there is no firmly established end goal. Because of the lack of a well defined end point, estimating costs and schedules is difficult.

2.1.5 Unified Process[2]

The Unified Process (UP) (developed by Rational) is an iterative incremental process with portions of all life cycle stages being performed on each iteration (requirements, implement, test, etc.). UP defines 4 phases of software development, with a milestone at the end of each phase to decide when to proceed to the next phase. Workflows are defined that extend across each phase. Workflows roughly correspond to the steps of the waterfall life cycle. Within each phase, several iterations are performed with aspects of all workflows performed in each. Each iteration is similar to a mini-waterfall, adding functionality with advancing iterations. UP embraces the use of Use Cases for requirements analysis.

The four phases are defined as follows:

1) Inception – The business case for the project is established. Initial use cases are defined. A baseline project plan is developed showing phases and iterations. Prototypes are developed.
2) Elaboration – The functional requirements are fleshed out. A baseline architecture is established. User manual is started. Further prototypes are developed.
3) Construction – A series of iterations adding functionality ending with the fully developed product and user manual.
4) Transition – Beta testing, training, and marketing.

The workflows consist of:

1) Business modeling – Ensure the software development dovetails with the business processes.
2) Requirements – Development of functional requirements of the system. Usually performed through the development of use cases.
3) Analysis & Design – Analyze requirements and develop software architecture.
4) Implementation – Code the requirements into the modules defined by the architecture. Integrate software modules into whole product.
5) Test - Ensure proper integration. Unit test modules. Ensure requirements are all implemented.
6) Deployment – Packaging, distribution, all activities associated with formal release of the product.

---

[2] Based on information from "Rational Unified Process – Best Practices for Software Development Teams" A Rational Software Corporation White Paper, © 1998, Rational Software Corporation

7) Project Management – Planning and resource management. Mitigate risk, monitor progress.
8) Configuration & Change Management – Ensure consistent configuration of product. Handle changes, defect reporting, multiple developers and simultaneous changes.
9) Environment – Set up and manage software development environment (tools, compilers, training, processes).

There are specific deliverables tied with each phase in the Unified Process, and each phase also has a milestone set of objectives the must be met to go on to the next phase.

- The Inception phase deliverables include a vision document, initial use cases, initial risk assessment, a project plan, and several prototypes. The Inception milestone is the Life Cycle Objectives which provides stakeholder concurrence on scope and schedule estimates, requirements understanding, credibility of cost and schedule estimates, depth/breadth of prototype, and actual expenditures versus planned expenditures.

- The Elaboration phase deliverables include a more complete use case model, any non-functional requirements have been captured, software architecture description, and development plan showing iterations. The Elaboration milestone is the Life Cycle Architecture which establishes if the project vision is stable, if the architecture is stable, if the plan is accurate, does the prototype show how major risks have been addressed, do the stakeholders agree with the vision, and is the resource expenditure against the project plan acceptable.

- The Construction phase deliverables include the software product integrated on the required platforms, the user manuals, and a description of the current release. The Construction milestone is the Initial Operational Capability which establishes if the product release is stable and ready for deployment, if the stakeholders are ready, and if the resource expenditure vs. plan is acceptable.

- The Transition phase deliverables include achieving user self-supportability, achieving stakeholder concurrence that all is complete and consistent with the vision, and achieving the final product baseline. The final milestone is the Product Release which establishes if the user is satisfied and if the resource expenditure is acceptable.

The Unified Process is closely tied with UML and object-oriented design, so may not be as directly applicable to structured approaches. Iterations and increments provide flexibility to add smaller portions of functionality and over time build up the full system. UP is well suited for management of change.

2.1.6 Life Cycle Choice

Each of the life cycles presented here have areas of strength, and areas of weakness. It is important for the Prometheus NRPCT software development effort to identify a software life cycle that provides the rigor necessary for safety critical reactor software, but is well tailored for the problem domain and does not invent steps merely for the sake of process.

2.1.6.1 Life Cycle Evaluation

Each software life cycle has concepts that it embodies that are more or less applicable to the development needs for the Space Reactor Module I&C system. The following table provides an evaluation of each life cycle presented above:

Life cycle	Positives	Negatives
Waterfall	1. Well defined. 2. Defined documentation. 3. Simple.	1. Doesn't react well to change. 2. Requirements must be well defined up front.
Spiral	1. Embodies risk management. 2. Use of prototyping.	1. High level understanding of all requirements needed up front.
Incremental	1. Allows for prioritized development and release of functionality. 2. Supports risk management to choose functionality developed per stage. 3. Promotes early integration.	1. Scope of requirements must be understood up front.
Evolutionary	1. Reacts well to change. 2. Use of prototyping.	1. Lack of well defined end point.
Unified Process	1. Reacts well to change. 2. Project management and metrics directly supported. 2. Use of prototyping.	1. Tends to be biased to Use Cases and object-oriented technologies.

Each life cycle identified above has valuable features necessary to the success of the Space I&C software development task, but also contains detrimental features as well. The incremental life cycle matches well with the Prometheus goals because many of the requirements will be understood prior to increment planning.

2.1.6.2 Life Cycle Choice - Incremental Life Cycle

The incremental software development life cycle was selected since it provides for the ability to separate the development into smaller increments to match up with the Prometheus integrated build schedule, it allows for the use of an iterative waterfall

approach within each increment, and risk analysis can be performed up front to aid in assigning functionality to each increment.

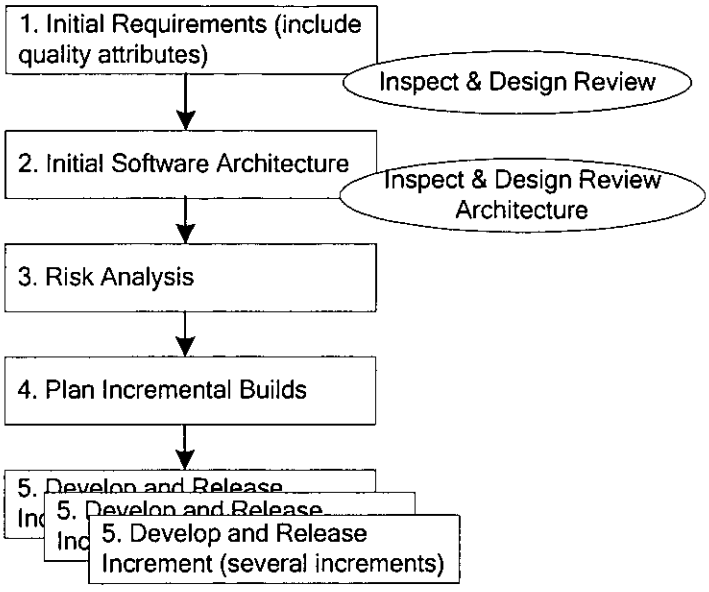
This life cycle uses an incremental development method (each increment is one pass through the "waterfall"). A single product release will contain at least one increment. However, the work necessary to generate a product release will normally be split into multiple, relatively independent increments. Each official release has progressively more features and capabilities than the previous release. These increments dovetail well with the JPL and NGST software delivery processes.

Each increment may contain several iterations as testing identifies deficiencies which may then drive changes to the requirements, design or source code. The iterations continue until all requirements for the life cycle task have successfully passed all testing. Based on lessons learned from the development and testing process, requirements may be relocated to another stage or may be removed entirely.

Within each increment, the various phases are presented much like the waterfall method, showing a flow from requirements, to implementation, to test. This allows for documentation of the requirements and the design since these items are critical for the Space I&C software development. Unit testing is shown prior to integration to reflect that each individual module should be tested prior to effort being expended in integration. The emphasis on team inspections and independent design reviews early in the life cycle will result in fewer defects detected in testing, where the cost to fix the defects is traditionally an order of magnitude greater. This also helps to minimize rework in later stages. One of the strengths of this model is the explicit representation of defect tracking and the ability to reenter earlier phases in response to defects. This may involve several iterations within a increment.

Figure 1 provides an illustration of the overall life cycle. Notice that the initial software requirements are split among the expected releases but that the final list of requirements is usually modified as a result of work performed during the individual stages.

Figure 1 – Overall Incremental Software Development Life Cycle



Initial Requirements – A high level survey of the functional requirements for planning and architectural purposes. In this stage a risk analysis can be performed to help determine which stage each requirement should be implemented in. These requirements are inspected and design reviewed prior to NR submittal.

Initial Architecture – An initial architecture is established at this stage to facilitate a framework for the staged delivery of functionality through the various versions. The architecture will be inspected and design reviewed.

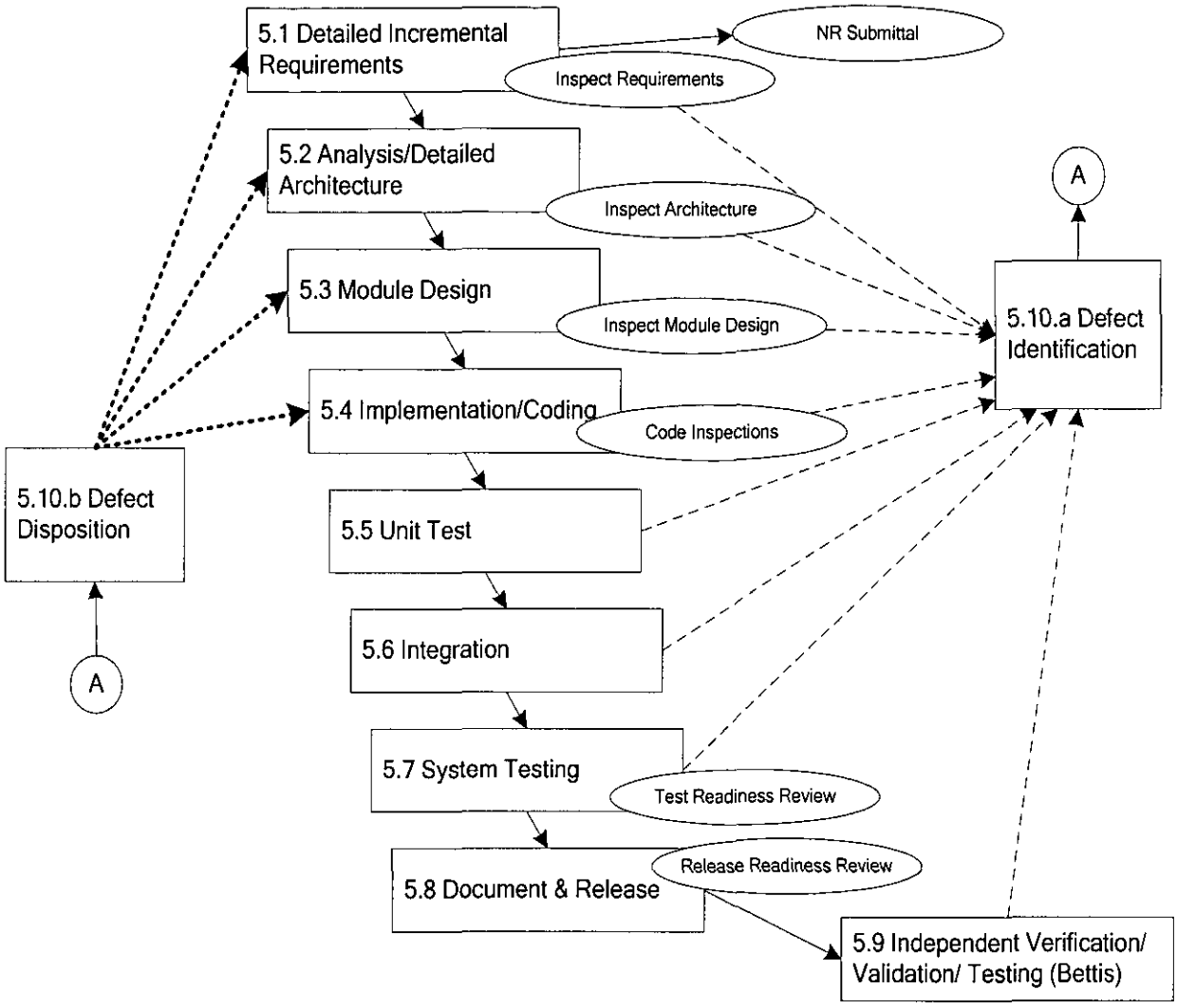
Risk Analysis – The risk analysis step allows for evaluation of the requirements and design to aid developing risk mitigation plans and assigning functionality to each increment.

Plan Incremental Builds – Each increment is planned out and functionality is apportioned to each build.

Increment – Within each Increment, the various steps of the software lifecycle are performed from design through implementation and test to provide the release version for that stage's functionality. A set of software, a requirements document of increasing fidelity, and architecture/implementation documentation are outputs of each design phase. Additionally, if necessary, risk analysis can be performed after each increment to determine if the initial splitting of requirements is still valid or if functionality should be shifted in following increments.

The following figure illustrates the process steps found in each increment.

Figure 2 – A typical stage of the Incremental Life Cycle



Requirements – This step includes a refinement of the functional requirements to be implemented in this increment. The requirements to be implemented were allocated as part of the initial requirements phase.

Analysis – During this step, the requirements are analyzed to ensure completeness and testability.

Design – Architecture is refined (initial architecture was established as part of initial requirements phase). Modules may be decomposed further. Architecture is documented.

Implementation – Software is written. This software shall be peer reviewed by other developers in the group to ensure consistent application of coding standards, or to get alternate ideas for implementation strategies.

Unit Testing – Individual modules are unit tested to ensure functional completeness and are also tested for MC/DC. These test cases are peer reviewed by other developers in the group. This is accomplished prior to integration to allow for timely identification of defects.

Integration – All modules are brought together and compiled to ensure the overall system will work.

System Testing – The integrated system is tested to ensure it meets the overall functional requirements.

Release – Once all previous steps have been completed and the software has reached the appropriate level of quality, the system will be released. This includes both the software, functional requirements document, and any design or implementation documentation.

Independent Verification and Validation – Determine whether the products for the increment fulfill the conditions imposed upon them. This includes the functional requirements, architecture and design, implementation and coding, testing procedures, and any associated documentation. May be conducted as an ongoing activity throughout the incremental development process. Develop and perform tests to determine whether the final software product fulfills the specific intended use.

Configuration Management – CM is applied to all stages of the process to ensure that requirements, design, code, and test cases are controlled. This allows for traceability and repeatability in the process. All artifacts subject to inspection, review, or test will be placed under CM prior to the start of the verification activity.

Deficiency – Identify - All deficiencies noted in artifacts under CM will be recorded and analyzed. This is applied at each step in the life cycle.

Deficiency – Disposition – Each deficiency must be evaluated to determine the proper fix, and the necessary rework to ensure proper regression testing.

This life cycle provides a robust framework to provide incremental releases as functionality is developed. It applies a rigorous process to ensure the high level of quality that is required by the NR program.

2.2 Design Methodology

Once a software life cycle has been determined there are different development methods that can be used to achieve the desired system functionality. Both the object-oriented methodology and structured methodology were considered for NRPCT software development. The structured methodology was selected as noted below.

2.2.1 Object-Oriented Methodology

The object-oriented methodology embodies seven principles in its approach to software design: abstraction, encapsulation, modularity, hierarchy (inheritance), typing (data types), concurrency, and persistence.[3,4]

Abstraction addresses complexity by considering only the properties of an object necessary in a particular usage that distinguish it from other objects. This is done to provide the most concise definition of the object that is possible. An abstraction of an object can be referred to as a class of objects.

Encapsulation involves combining both data and the operations (functions) performed on that data as tightly coupled. Through data hiding, an interface is defined for an object that only exposes data or operations that are required by users of the object. This allows a separation between the internal implementation of an object and the external interface. This data hiding helps to make a concise design and prevent errors involved with unintended changes.

Modularity involves breaking a problem into smaller self-contained chunks (modules), and minimizing the interfaces between these modules. This allows for a logical view of the system, and helps to promote modifiability since the scope of changes can be minimized to the module level.

Hierarchy involves ordering classes of objects through inheritance. This allows sub-classes of objects to inherit the interface and functionality of a parent class, while refining or adding more specific functionality as part of the sub-class (child class). This allows for class hierarchy trees with potentially many layers. Inheritance may be through single inheritance, where a child class may have only one parent (base) class, or it may be through multiple inheritance, where a child class may have several parent classes. Single inheritance is the clearest to understand, while multiple inheritance may lead to confusion and great care is needed to avoid errors.

Typing enforces the class of an object so that different objects may not be casually interchanged. Strong typing ensures that objects can only be treated by their class, weak typing allows for simple conversions between different classes, static typing is bound at compile time, while dynamic typing is evaluated at run time. Related to typing is the concept of polymorphism, or "many forms". An example of polymorphism is the C++ virtual functions which allows a base class to define a virtual function which is then implemented in two or more separate child classes. When the child class objects are treated as a base class object, and the virtual function is called, the child class implementation for that function is called.

Concurrency is the process of running several actions or processes at the same time.

Persistence refers to the continued existence of an object after its creator has been destroyed.

---

[3] Information taken from Object-Oriented Analysis and Design, by Grady Booch, © 1994, Addison-Wesley.

[4] Methodology information from Object-Oriented Technology (OOT) In Civil Aviation Projects: Certification Concerns (1999), by Leanna K. Rierson, FAA 1999

The Object-Oriented methodology includes several phases, Object-Oriented Analysis, Object-Oriented Design, Object-Oriented Programming, and Object-Oriented Verification and Test.

Object-Oriented Analysis involves defining all of the classes necessary to solve a particular problem, and the behaviors and relationships of those classes. Several models are used to identify all of these classes, including use cases, class-responsibility-relationship (CRC) models, object-relationship models, and object-behavior models. Use cases allow identification of requirements, CRC is used to identify the classes and hierarchy, object-relationship helps to identify the relationships between various objects, and the object-behavior helps to determine the necessary behaviors of each object.

Once the Analysis has been completed, Object-Oriented Design is used to translate the identified classes into a software architecture. This is performed through four layers of design. The subsystem design layer separates the system into various subsystems necessary to achieve the functionality. The class and object design layer separates each subsystem into class hierarchies. The message design layer defines the communications between objects. The responsibilities design layer deals with individual algorithm design and data structure design for each object.

Object-Oriented Programming is used to implement the design using an object oriented language such as C++ or Java.

Object-Oriented Verification and Test involves reviews, analysis, and testing of the software. Testing has to be able to verify features of the object-oriented design, such as encapsulation or any polymorphism used.

The Object-Oriented methodology embodies a view of system function and design that is significantly different than the traditional structured approach. The use of classes and inheritance provides valuable features, but also create new concerns for verification and validation.

2.2.2 Structured Methodology

Originally developed in the 1970's the structured method sought to improve programming techniques through the use of functional decomposition. The goal of structured programming was to improve programmer effectiveness and decrease the error rates over the traditional monolithic 'spaghetti code' style of programming. This is accomplished primarily by decreasing the reliance on GOTO statements by providing conditional constructs. Structured programming has three central concepts: Top-down development; Modular design; and The Structure theorem.

Top-down development seeks to break down the application into manageable pieces using functional decomposition. This is done by outlining a general solution then systematically breaking it down into detailed steps. This process is continued iteratively until the details are fully flushed out.

Modular design is an extension to top-down development in which related tasks are grouped together. By grouping similar functions together readability increases and it becomes easier to understand the system. The increased understanding and modular design make maintenance and the adaptation of new functionality easier.

The Structure Theorem states:

> "It is possible to write any computer program by using only three basic control structures:
>
> - sequence;
> - selection, or IF-THEN-ELSE; and
> - *repetition, or DOWHILE (or simply WHILE).*"

In pure structured programming it is recommended that each loop, and function, have only one entry point and one exit point. There are certain cases where it is impractical to follow this and it is typically not enforced at the compiler level. In the general case however, this should be striven for in order to increase readability and reduce the chances of going down an unintended code path.

In structured programming modularization is accomplished by decomposing program algorithms into subalgorithms (typically called functions or procedures). These functions can themselves be broken down further. Unlike object-oriented methodology there is no fundamental relationship between data and behavior in the structured methodology. This means that the association of data and its behavior must be controlled by the program itself. Typically this is done by passing data to subprograms via arguments and parameters.

In the analysis phase graphic models are typically used to specify context, process, and control. The context deals with inputs, outputs, and their sources. Process focuses on the functional behavior of the procedures, their interactions and relationships to the inputs and outputs. Control addresses the issue of under what circumstances each of the functions is performed.

In the design phase a graphic model of the system is created. This model is used to identify tasks, define task interfaces, develop preliminary software architecture, decompose tasks, and define the data dictionary elements.

Some of the graphic models typically used are:

Context Diagram – Shows external interfaces to the software/module/function

Data Flow Diagram (DFD) – Shows the major decomposition of functions and their interfaces. Typically they are used to follow the path of the data as it moves through the system.

Task Communication Graph (TCG) – Provides a visual representation of concurrent task and their interfaces.

Software Architecture Diagram (SAD) – Identifies the grouping of tasks on the TCG.

Structure Chart – Defines the partition of the elements shown in the SAD into a hierarchy.

Data Dictionary – While not strictly a graphic model, the data dictionary is used in conjunction with the other diagrams to define the individual entities.

2.2.3 Method Choice - Structured

Both structured design and object-oriented design provide a disciplined method for effectively designing a software implementation. Both methods are widely used in industry for software development and have had systems successfully created based on the method's embodied principles.

Structured design methodology has been chosen for the Prometheus project NRPCT software development. The choice of structured programming over object-oriented programming was driven by several factors: 1) Structured programming has been used far more frequently in space embedded applications, 2) There are many aspects of object-oriented design that must be avoided or closely monitored to ensure a robust real-time embedded design (e.g. polymorphism (virtual functions)), 3) There is a larger body of experience with structured design outside of the programming community, and this experience can be leveraged to aid with design reviews and inspections.

Object-oriented (OO) technology is very capable of providing robust software designs. Many of the fundamental principles of OO such as encapsulation, hierarchy, and modularity are very powerful techniques to manage complexity and help to minimize interfaces between modules. These all help with maintainability of the software. The downside of OO for real-time embedded systems comes with dynamic memory allocation, confusion that can arise with polymorphism, run-time type information (typing), and certain aspects related to abstraction (templates). Many of these features have proven themselves to be either error prone, or may make it complicated to prove time response requirements can be met. It is possible to overcome these concerns with coding standards and careful inspections.

Structured programming on the other hand, does not have the built-in support for data hiding and class hierarchies like OO, but a well thought out functional decomposition and data flow diagrams can go a long way to mitigating these concerns. Similar to OO, structured programming requires discipline and care when design and implementation are performed. No method or programming language itself will render a perfect product, but careful design along with appropriate review, inspections, and testing will maximize the potential for a high quality product. The greater experience with structured methodologies, and the reduced set of potential embedded programming pitfalls combine to make structured programming the method of choice for NRPCT Reactor I&C software development.

2.3 Language

Once a programming methodology has been chosen, and certain details are understood about the software architecture, a programming language can be selected to meet the needs of the software project. There are many different language choices available; all have advantages and disadvantages depending on the application domain. Language examples include C, C++, Assembly, FORTRAN, COBOL, Ada, BASIC, and others.

2.3.1 C

The C language is a third generation, or high level language that has been in use for several decades. C was developed for systems programming but is well suited for general programming as well. C is designed for a structured representation of system functionality, in that it works by function call and return.

2.3.2 C++

The C++ language is an extension of the C language to embody the principles of object-oriented programming. C++ allows the definition of classes, where both data structures and the functions that use the data are bound together in one definition. Many of the principles of object-oriented methodology are directly implemented in the language, including hierarchy (inheritance), abstraction, encapsulation, and typing (modularity can be achieved in either C or C++). While C++ directly supports object-oriented programming, some features of C++ can be of concern for safety critical systems. These features include dynamic memory allocation, multiple inheritance, polymorphism (through virtual functions), templates, exception handling, and others.

2.3.3 Assembly

Assembly language is a second generation, or low level language. It is a mnemonic representation of machine code with symbolic values for variables and address offsets. Assembly language can be very powerful for fast execution and hardware access, but is very platform dependent and so limits the portability of what is developed. Assembly language may find its greatest use in various board support or hardware driver code.

2.3.4 Other

Java is similar to C++, but makes use of a virtual machine for interpretation. Java is not a likely candidate due to inherent unpredictability with the garbage collection memory management features. FORTRAN is a structured language that has found great use in numeric intensive applications, but it is not well suited for embedded programming. COBOL and BASIC are both languages that are used primarily for business oriented applications, and BASIC is usually an interpreted language. Ada has found some use in space applications and also has a strong history with embedded military applications; however, many of the Prometheus software development organizations have stronger experience with C and C++ and Ada does not have as large of a developer base.

2.3.5 Final Choice - C

With the choice of structured design, the C language becomes a natural choice for software implementation. C is widely understood and recognized. C has been standardized by ANSI and has been used for decades in both embedded and general programming situations. Compilers are readily available for C, so the toolset is easy to acquire. When necessary, it is easy to incorporate assembly level modules for interfacing directly with the system hardware (registers, I/O) with modules coded in the C language.

3 CONCLUSION

After careful evaluation of life cycles, methodologies, and languages, the NRPCT has chosen an incremental life cycle, a structured design methodology, and the C programming language for Prometheus reactor I&C software development. It is felt that

these decisions support not only the JIMO type mission, but are also very extensible to other reactor concepts, including potential lunar surface missions.

NRPCT Reactor I&C Software Development

Process Manual

(NRPCT-RIC-SDPM-001)

NRPCT Space Electrical I&C Software Team

July 2005

Revision History

Revision	Author	Date	Change Synopsis	Reason for Change
Draft	Thomas Hamilton	7/10/05	Original	

Table of Contents

1 INTRODUCTION

1.1 Identification

This is the NRPCT Software Development Process Manual (SDPM) for the Prometheus project.

1.2 Purpose

The SDPM establishes the software development, and verification and validation processes for the NRPCT Space I&C software development for project Prometheus. This work includes but is not limited to the flight software, ground telemetry and analysis software, test beds, and vendor developed sensor and actuator interface software. These practices exist to ensure that the I&C software is of sufficient quality to meet the Prometheus needs, particularly the paramount need of safety.

1.3 Basis and Standards

This document expands upon and traces to the Prometheus Software Management Plan (PSMP), JPL document #982-00046. The PSMP defines the high level software requirements and processes to be used throughout project Prometheus. Although the Memorandum of Understanding and the Memorandum of Agreement between NASA and Naval Reactors do not give JPL approval authority over NRPCT software development, the NRPCT desires to maintain as much commonality in software processes as possible with the rest of the Prometheus team. Following the principles established in the PSMP helps to maintain commonality among software developed by various Prometheus software development organizations.

This document incorporates guidance from the Prometheus Software Quality Assurance Requirements (JPL Document #982-00038) to ensure that there is commonality with software qualification across the Prometheus project.

This document incorporates guidance from several NRPCT standards and policies. The SDPM incorporates guidance from the NR Software Engineering Policy (as documented in Bettis Letter No. B-REO(M)CD-008, 3/16/05). The SDPM requires that the software quality criticality level (SQCL) for the flight and ground software be determined per the NNPP Standard for Software Qualification by Criticality Level (as issued for three prime concurrence by KAPL Letter No. ARP-68640-0305, 9/2/04).

1.4 Document Hierarchy

The Software Development Process Manual provides the process definition to be used for NRPCT Prometheus software development. The SDPM expands upon the guidance of the PSMP. The SDPM is the highest level document in the NRPCT software development documentation structure. Lower level documents tracing to the SDPM will cover the definition of coding and design standards and checklists, document templates, and mission specific Software Development Plans (SDPs) as follows:

- The NRPCT Software Standards (NRPCT-SW-STDS-001), which provide standards and checklists for requirements development, design documentation, and coding standards.
- The NRPCT Software Templates (NRPCT-SW-TMPL-001), which provide documentation templates for requirements documents, test reports, test plans, and other work products that have multiple instances to allow for a common look and feel.
- The NRPCT Software Development Plan (NRPCT-XX-SDP-001, where the 'XX' vary from mission to mission as a unique identifier), which provides high level definitions for various lower level mission specific software development plans. The NRPCT SDP is mission specific, and defines the overall software development plan for the specific mission. Following the NRPCT SDP are subordinate SDPs, and under these subordinate SDPs are work breakdown structures and schedules.

1.5 Definitions

Baseline – A set of items under configuration control such that each individual revision level is captured and treated as a collection with its own unique identification. The baseline for the collection can always be returned to even after changes and further baselines have been created.

Context Diagram – Overall graphical representation of software modules to show relationships to each other in a functional hierarchy.

Dataflow Diagram – Representation of the flow of data between software modules in a software architecture, also identifies inputs and outputs.

View – From a software architectural standpoint, a view is a representation of a software architecture used to communicate certain information about the architecture to a group of stakeholders (e.g. a functional decomposition view would show developers the software modules and the hierarchical relationships between them).

1.6 Acronyms

Table 1

Acronym	Definition
CTD	Composite Test Device
DSS	Deep Space System
I&C	Instrumentation and Control
FRD	Functional Requirements Document
IEEE	Institute of Electrical and Electronics Engineers
JPL	NASA Jet Propulsion Laboratory
MC/DC	Modified Condition / Decision Coverage
NRPCT	Naval Reactors Prime Contractor Team
NSDP	NRPCT Software Development Plan (this document)
PCAD	Power Control and Distribution
PSMP	Project Software Management Plan
PSR	Project Software Requirements

SDPM	Software Development Process Manual
SDVP	Software Development and Verification Platform
SHA	Software Hazard Analysis
SFTA	Software Fault Tree Analysis
SFMECA	Software Failure Modes, Effects and Criticality Analysis
UP	Unified Process (Rational Unified Process)
V&V	Verification and Validation

1.7 References

N/A

2 SOFTWARE DEVELOPMENT PROCESS

The software life cycle chosen for NRPCT Space I&C software development consists of the incremental life cycle. This life cycle imposes a specific sequence of events on the overall development of software. Each event has a process associated with it. Reviews provide an important quality gate to transfer to from step to step. These processes with the reviews, in addition to the ancillary processes are identified as follows:

2.1 Overview

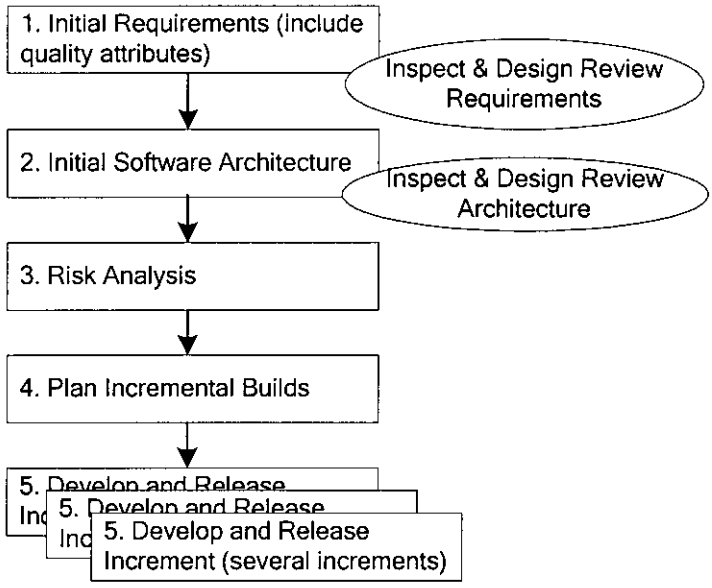


Figure 3: Software Development Process Layout

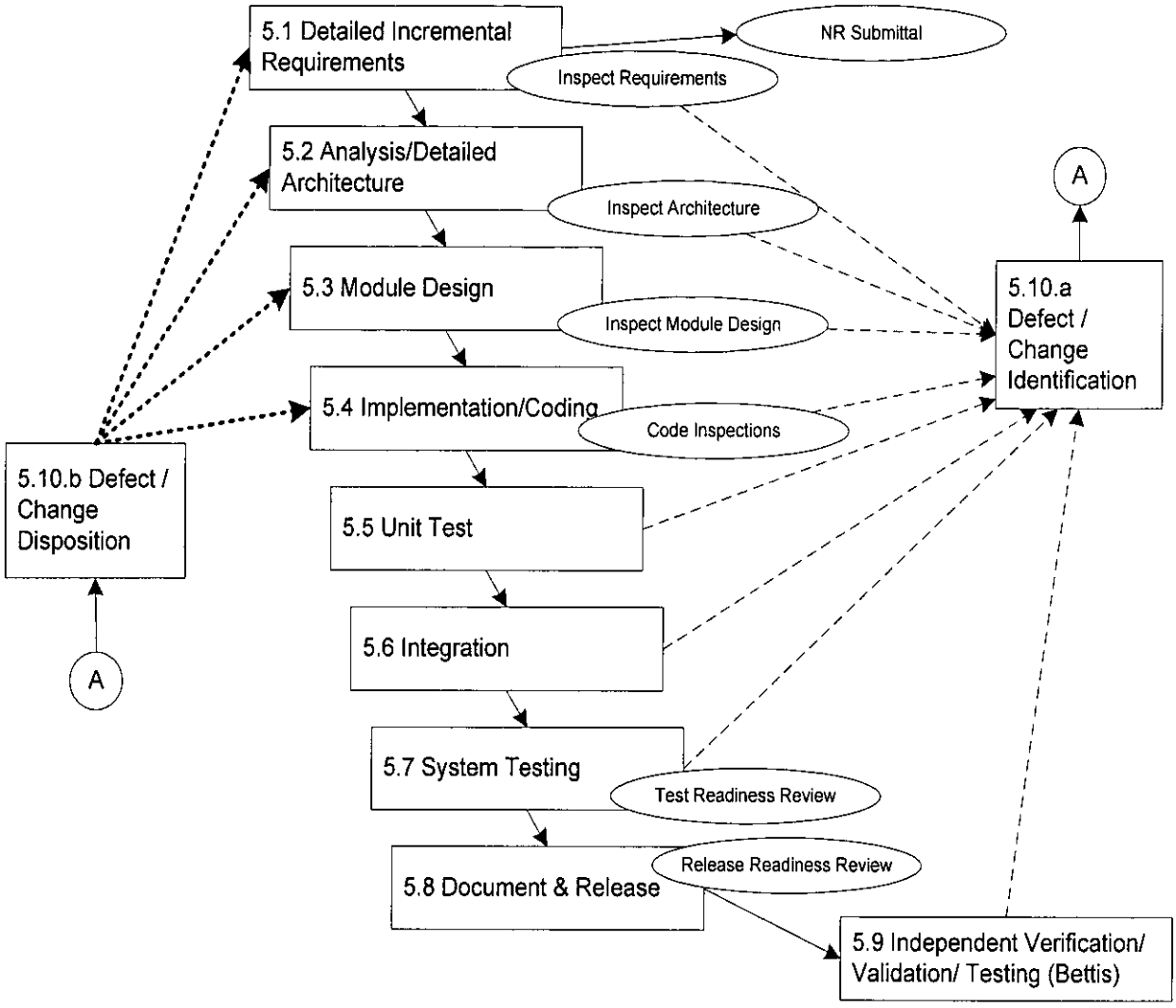


Figure 4: Detail of Task 5 (Develop and Release Increments)

Figure 3 shows the detailed layout of the incremental life cycle. Figure 4 shows the detail of the "Develop and Release Increments" step of the life cycle. Each increment is developed as an iterative waterfall.

2.2 Software Development Tasks

Each software development task associated with the incremental life cycle has a process associated with it, work products that are developed, and support processes that are used concurrently with the task.

2.2.1 Task 1: Initial Requirements

Before software requirements can be established, a high level overview of the system architecture needs to have been established. This provides a framework for the

functional division of responsibility between hardware and software, and also the allocation of functions between modules and tiers of instrumentation and control. This information provides the basis for the software functional requirements. Software requirements are initially derived from overall system requirements and any design constraints placed on the system.

There are many other constraints and sources of requirements that need to be included in the software functional requirements. Many of these items include quality attributes, maintainability, reliability, availability, security, etc. Some of these requirements are defined by JPL or NRPCT established guidelines. Collectively, these provide a source of software requirements that both drive functionality (e.g. fault tolerance and reporting), and software architecture (e.g. maintainability requirements). Interface requirements imposed by other portions of the system shall also be considered in this phase, including communications requirements and parameters.

The Prometheus project may require the development of separate (but similar) versions of the Reactor I&C software to support the I&C systems, including several phases such as test beds, or even prototype reactors. Development of subsequent versions will make heavy use of functionality defined in the initial development cycle. Making use of common functionality between the different software versions ensures continuity in the various software designs. Identification of the commonality also allows changes to requirements to be evaluated across all platforms in a more efficient manner.

As software requirements are developed, they are entered into a requirements management database, which is under configuration control, allowing future traceability to implementation and test. Placing the initial requirements in the requirements management database establishes an upfront framework to allow for clear tracking and review of the software functionality.

Once the initial software requirements have been captured, they are inspected to validate the requirements to the intended functionality. This inspection is a formal process involving members of NRPCT, but may involve other Prometheus team members as appropriate. The initial software requirements are not expected to be complete at this stage since this early in the development cycle there is a great deal of uncertainty in the overall system architecture and system requirements. There is also uncertainty as to the target hardware and sensor suite. These areas of uncertainty will make it difficult to completely specify all of the software functionality; however, there are many functions that have a high level of confidence. At the initial requirements phase, as much detail should be captured as possible.

If there is sufficient confidence in the requirements at this point, the software requirements will proceed to undergo a formal design review. The design review is a formal process involving independent engineers not directly involved in the work product. This review involves a broader spectrum of people than the inspections.

Once the requirements have been captured and inspected, any changes from the design review are then rolled back into the requirements. Once this has been accomplished, work can proceed to the next task.

2.2.2 Task 2: Initial Software Architecture

In order to successfully develop the initial software architecture, a fairly complete set of initial requirements must exist. While the requirements may not be fully specified to the lowest level of detail, the requirements should cover the overall scope of the system to allow identification of all necessary software modules. The design methodology used for Reactor I&C software is structured design with the associated top-down design approach and functional decomposition.

As noted in the Initial Requirements development task, there is a great deal of common functionality expected to be identified between the various versions of software that will be developed as part of Prometheus. The development of these versions will be very close in time, this requires a great deal of commonality in the software design and implementation effort. Since there will be a common core of functionality, the software architecture can be generalized to allow as much software reuse as possible between platforms. Software reuse is imperative to ensure the aggressive Prometheus schedule can be attained.

Structured development requires the use of a top-down development strategy. This starts with considering the full system, and then splitting the functionality in a hierarchical manner down to the module level. From a software architecture standpoint, there will also be common functionality between I&C system components (MOL structure, fault management, etc.). This common functionality shall be identified in the software architecture to allow sharing of as much structure as possible between system tiers.

Once the functionality has been decomposed to the module level, the data structures shall be defined, and the data flow identified. This establishes the interface between the various software modules, and attention can be given to ensuring the integrity of the data.

The initial architecture is documented in a manner to allow clear communication of the design for the developers, for peer reviewers, and for the Customer. Attention also needs to be given to traceability to ensure that the requirements will be traceable to the implementation. Often software architectures are documented through several views, where each view highlights different aspects of the architecture. For example, context diagrams show how the modules are functionally related to each other, state diagrams show the transitions between various states in a state machine, and data flow diagrams show the run-time flow of data through the system with the operations performed on the data.

Once the initial architecture has been developed, an inspection shall be performed on the architecture to determine its suitability for the various requirements. Similar to the software requirements, the software architecture shall also undergo a formal design review to ensure a strong foundation has been laid for the various I&C software components.

2.2.3 Task 3: Risk Analysis

Upon completion of a set of initial functional requirements and an initial software architecture, the requirements and the architectural elements shall be evaluated from a risk standpoint. There are many different types of risk involved in the Prometheus

project, from risks that may cause system failure, to technical unknowns that may constitute a schedule risk.

Risks for the requirements and architecture shall be identified. Once the risks have been identified, the level for each risk shall be assigned, and mitigation strategies shall be determined to aid in eliminating or minimizing the risk. Once the risks have been identified, they shall be formally documented to ensure that the risks can be referred to later in the project. The stakeholders involved in identifying and evaluating the risks shall include the software developer, system engineer, and others as appropriate. Once the risks have been identified and mitigation strategies established, these shall be monitored through the lifecycle to ensure proper closure of the risks according to plan.

A Software Hazard Analysis (SHA), Software Fault Tree Analysis (SFTA), and a Software Failure Modes, Effects, and Criticality Analysis (SFMECA) shall also be performed to further examine the requirements and architecture. These are tools that help to provide a different point of view to further examine potential weaknesses for addressing.

2.2.4 Task 4: Plan Incremental Builds

Incremental development is ingrained in the overall Prometheus software development process. The logic behind incremental development is to allow compiling, inspection, testing, and delivery of portions of the overall system functionality in phases to develop confidence in the project and to build upon past successes. In this manner, risk can be spread over several increments and managed better than having one large system release.

Planning the increments consists of determining what system functionality and which software modules will be delivered in each increment. Also it involves ensuring that later increments build upon the functionality of earlier increments.

Certain essential functionality must be developed to allow the design of a workable set of code. This includes setting up the Main Operating Loop (MOL) and basic system interface components. Functions with identified risks should also be addressed in the early increments to provide more time to test the requirement to ensure that the requirement is fully understood and the implementation is satisfactory. Later increments can contain basic control functions, leading up to full autonomous control with full fault management and telemetry.

As part of the increment planning task, a Work Breakdown Structure (WBS) shall be developed (or at least enhanced since one should already exist to help guide the initial steps) based on the initial architecture software module definitions. As functions are then allocated to each increment, the WBS task items corresponding to software modules naturally follow this incremental allocation. Schedules shall then be laid out with the delivery dates for each increment aligned as closely as possible with higher level project need dates. The schedules shall then have developers identified to perform each task, as much as is practical. The basic development schedule for each increment shall follow the iterative waterfall defined for the life cycle, allowing margin for some iteration as defects are identified. If larger defects are discovered that significantly upset the architecture or require a great deal of rework, the plan will need to be reevaluated to

ensure that the schedule is realistic and can be achieved. Determination of the incremental release target dates and number of increments may be driven by overall Prometheus project needs, and additional increments may also be released to facilitate Independent Verification and Validation.

2.2.5 Task 5: Develop & Release Increments

This task is repeated as necessary for each increment defined in Task 4. This task is made up of several steps defining the waterfall portion of the life cycle. Each step has quality activities associated with it. Verification and validation activities are also part of the tasks, including code inspections, unit testing, integration, and system testing. The increment plan and risk mitigation plan shall be evaluated by the development team through the performance of this task for update if strategies change, or defects encountered change the basic assumptions for task sequence or duration.

2.2.5.1 Task 5.1: Detailed Incremental Requirements

This step focuses on the functional requirements allocated to modules to be implemented in the current increment. The requirements are refined to a sufficient level of detail that they are able to be implemented and tested. As requirements are refined, any emergent information on the system shall be taken into account and the software requirements must be fully reconciled with the system requirements and any other self imposed or derived requirements. If information is available to correct or refine the requirements not included in this increment, the requirements may also be refined. It is important that the whole set of software requirements be self-consistent, and not have portions that are incorrect.

A Communication Specification (Interface Control Document/Interface Requirements Document) shall be developed (or updated) at this time to define the parameters, ranges, defaults, and communication pathways for the system. Focus should be on functionality necessary for this increment, but care should be given to establish inter-module interfaces early in the project.

Once refined, the requirements shall be inspected, design reviewed - if not already design reviewed prior to increment planning - then submitted to NR for approval. If the refined requirements affect interfaces outside of the Reactor I&C, then concurrence shall be sought from the affected stakeholders. NR approval is not required to begin working on the following steps, but is required prior to release of the increment software.

2.2.5.2 Task 5.2: Analysis/Detailed Architecture

Once the requirements have been refined, they shall be analyzed to see if there are any architectural updates necessary in the light of new information that may come from the updated requirements. Architectural updates in later increments should be very carefully considered since there is a potential to impact already developed and tested software. Late architecture changes come with a high price tag in rework and requalification. Changes to the architecture can have effects on both software reuse, and various fault tolerance and risk mitigation strategies that have been built into the architecture.

For the focus areas of the current increment, there may be a need to add further detail to the overall architecture. These architectural updates shall be inspected to ensure that no incompatibilities are introduced in areas where the architecture supports multiple

systems or platforms. If a design review has not yet been performed prior to increment planning, a design review shall be performed prior to entering into module design.

2.2.5.3 Task 5.3: Module Design

At this point, it is necessary to lay out the internal functionality of the individual module. There shall be traceability between the module and the functional requirements implemented within the module. Internal functions and data types shall be defined, along with how the data is used. Fine detail shall be provided for functions that are used by other modules. An inspection shall then be performed on the module design.

2.2.5.4 Task 5.4: Implementation/Coding

Source code is developed for the module following proper coding standards. The source code shall be inspected by other software developers to ensure that the system functionality has been achieved and that the module adheres to the accepted coding standards. There shall be traceability established from the source code back to the functional requirements in a traceability matrix. Code must be compiled and may be desk checked (run with simple test cases informally), and any available static analysis tools shall be employed to help minimize the number of defects that may pass through to later inspections and testing. This ensures that the code meets minimum standards for usage so that it may be unit tested. Code shall be placed under configuration control to ensure work is not lost prior to inspections and unit testing.

2.2.5.5 Task 5.5: Unit Test

The cognizant software developer shall write unit test cases for the module(s) he or she developed. The unit test cases shall cover both requirements based testing (black box) and structural testing with modified condition/decision coverage (MC/DC) coverage (white box testing). The unit test cases shall be peer reviewed by a different developer to ensure coverage and adequacy. The test cases shall be run against the module, and any failures shall be corrected prior to releasing the module to integration. The unit test cases and test results shall be archived along with the source code and any associated test code (test harness) in configuration control. A unit testing report shall be issued to document the results of testing along with defects and other appropriate metrics.

2.2.5.6 Task 5.6: Integration

Once all of the modules for an increment have been developed, the various modules shall be integrated together and run on the target hardware. Any test cases developed to verify the integration shall be placed in configuration control and reviewed. This is a necessary step to releasing the final software, and also may catch incompatibilities between modules. Any defects detected during integration shall be placed in the defect tracking system for disposition. The generated object code and final linked executable shall be placed under configuration control, baselined, and released to system testing.

2.2.5.7 Task 5.7: System Testing

System testing is performed on the integrated software on the target hardware. System testing is performed with the aid of a Composite Test Device (CTD) that simulates reactor and plant behavior to allow testing of the control algorithms and telemetry feedback. System testing covers both functional testing, and also structural testing with MC/DC coverage for as much code as can be reached when fully integrated. Some

code may not be reachable at the system level without destructive testing, such as some fault detection routines. Unreachable code shall be inspected for suitability and removed if not necessary; it also should have been 100% unit tested. Any defects detected during system testing are placed in the defect tracking system for disposition. System test cases should be automated to the greatest extent possible to allow for rapid regression testing when iterating through the waterfall once defects are corrected. System test cases and results shall be placed under configuration control and be traceable back to the software module and functional requirement. A Test Readiness Review is conducted prior to system testing to ensure that the configuration of both the test setup and the software is correct, and that the software and test cases are ready for testing.

2.2.5.8 Task 5.8: Document & Release

Once all software inspections, unit testing, and system testing have been satisfactorily completed, the functional requirements have been approved by NR, and the software and executable have been placed in configuration control and identified with version numbers the baselined software may be released. All source code, object code, and executables shall have a revision list provided showing the version of code for each source file present in the build. Changes shall be identified from the previous released baseline. All traceability matrices should be complete. All identified defects should have been resolved or documented and provided to be fixed in next release if appropriate (every effort should be made to resolve defects prior to release of the iteration). The Release Readiness Review is performed to ensure that all of these necessary tasks have been completed. Once the work products have been approved, they are released with the appropriate documentation and archiving.

2.2.5.9 Task 5.9: Independent Verification/Validation/Testing (Bettis)

Once the software has been released for the increment, it is sent to Bettis for independent verification, validation, and testing. All of the qualification documentation is made available for review, but Bettis will also independently develop test cases for qualification of the software based on software and system requirements. Any defect detected by Bettis will be entered into the defect tracking database for disposition. Test cases and test harness used by Bettis will be placed under configuration control at Bettis.

2.2.5.10 Task 5.10a: Defect/Change Identification

Once a defect or change has been detected in any phase of the process, it shall be logged into the defect tracking database. This database may also be used to document change requests not directly tied to defects. Identification of a defect needs to clearly describe relevant information, such as the inputs, system state, and effects of the defect. The identifier should make an effort to reproduce the defect and note the sequence of events necessary for duplication. All of this information is critical to being able to correctly analyze the defect and create a proper disposition.

2.2.5.11 Task 5.10b: Defect/Change Disposition

Once the defect has been entered into the defect tracking database, it is necessary for the developer to evaluate the defect and come up with a recommended resolution. Correction of the defect is dependent upon the phase of the project. If the increment is

in the coding phase, the developer may simply fix the module and close out the defect report. Later stages require higher levels of approval to fix and appropriate retesting to ensure the fix is complete and has not affected other functionality.

For defects discovered in early phases up to and including integration, the integrator and developers are the prime stakeholders. For defects involving functional requirements, NR becomes a stakeholder since they are the approval body for any requirements changes. For defects discovered in system testing, the testers become additional stakeholders, and for software that has been formally released, NR, other members of the NRPCT, and even other Prometheus organizations may be stakeholders. The CCB determines the final disposition of a defect and when it will be corrected in the software release. Once a defect has been identified and corrected, it must be successfully retested to close out the defect report. Change disposition is subject to a similar process and requires CCB approval to be incorporated into the software.

Some defects may have a larger scope than a simple code fix. It may be necessary to change the software architecture, or even the functional requirements. Functional requirement changes would require a resubmittal of the requirements to NR. Requirement traceability will facilitate determining which modules must change and what test cases would need to be updated and rerun on the affected modules.

2.3 Ancillary Activities

These processes apply across multiple tasks and provide the support structure for the overall software development life cycle.

2.3.1 Configuration Management

Configuration management combines aspects of both configuration control and change management. Configuration control ensures that a particular version of a work product at any given time is known, and previous versions can be recovered if necessary. Configuration control also coordinates simultaneous update of software products and multiple active versions of software products. Change management ensures that changes to the various work products are reviewed and managed to ensure the needs of the project are achieved. Various work products may have different methods for configuration control. Functional requirements are controlled through a requirements database (e.g. Cradle). This provides a framework for controlled checkin/checkout, baselining, and auditing. It also provides facilities for requirements traceability. Source code, executables, test cases, test results, and relevant design and analysis documentation are stored in a configuration management database as well. This database may not be the same database used for managing functional requirements. The same requirements for controlled checkin/checkout, baselining, and auditing exist for these work products. Configuration management is closely tied with the defect disposition process in that the Configuration Control Board is the governing body that determines when to modify a software item. The Configuration Control Board (CCB) is comprised of various stakeholders in the software development process, and expands in scope as the number of stakeholders grows.

2.3.2 Requirements Traceability

Bidirectional requirements traceability is critical to the success of Prometheus software development. Bidirectional traceability means that each requirement must be traced to

the design and a source code module implementing it, but it must also be possible to take a source module and identify which requirements it implements as appropriate for the identified software criticality level. Additionally, test cases must be traced to the module and requirements. This allows for full through-process evaluation of changes to ensure the full impact may be understood. There are many tools that facilitate traceability.

2.3.3 Defect Tracking

Defects shall be placed into the defect tracking system at the various stages of the life cycle. Each defect shall be reviewed and a disposition assigned for proper closeout of the defect. This may include revision of a software item, requirement, or the architecture. Once an item has been revised, all of the affected quality steps must be repeated for the scope of the change. Refer to life cycle tasks 5.10a and 5.10b for further discussion on defect tracking.

2.3.4 Inspections

Inspections are formal reviews performed by a group of the developer's peers and will be performed as defined by IEEE Std-1028. An inspection ensures correctness and proper direction prior to more formal stages of testing and review. For requirement reviews, the peers consist of other developers and system engineers. For software architecture reviews, peers consist of developers or system engineers knowledgeable of both software and the system design. For code inspections, the peers consist of other software developers. The general results of a peer review shall be documented in a letter to the software manager. Engineers from other portions of the NRPCT or other Prometheus development organizations may be included in the inspections as appropriate.

2.3.5 Design Review

Design reviews are formal reviews performed in accordance with the KAPL Quality Assurance manual (KQA-10), and make use of a group of independent reviewers outside of the immediate group. These reviewers may consist of engineers from KAPL, BPMI, Bettis, and members from other Prometheus development organizations. The design review ensures the adequacy of the work product. Design reviews are required for functional requirements documents prior to submittal to NR, and for the software architecture once it has been refined to a sufficient level of detail.

2.3.6 Test Readiness Review

The Test Readiness Review is used to ensure that all of the previous quality activities for software have been completed and that the test plan is adequate for the components under test. This requires the configuration for both the software and the test environment to be known. There may be several readiness reviews during the full course of system testing as different aspects are tested to ensure the scope of the review is manageable. This also requires the development of adequate test cases to qualify the software. Members of the Test Readiness Review include the software developers, testers, system engineers, and lead engineers.

2.3.7 Release Readiness Review

The Release Readiness Review is used to ensure that all of the quality activities have been completed prior to releasing the software for use by Bettis for Independent V&V, or by other Prometheus development organizations. Members of the Release Readiness Review include the developers, testers, system engineers, managers, and NR. The Release Readiness Review Report will identify all qualification activities and dates and document the exact configuration of the release.

2.3.8 Software Hazard Analysis, Software Fault Tree, and SFMECA [RESERVED]

A software hazard analysis, software fault tree analysis, and a software failure mode effects and criticality analysis shall be performed on the software for each version. Any defects identified in these analyses shall be placed in the defect tracking database.

2.3.9 Auditing/Self-Assessments

Self-assessments shall be periodically performed to ensure compliance to the processes established by the SDP. Audits shall be performed by the KAPL SQA group to provide an independent verification of compliance to the SDP.

2.3.10 Software Criticality Level Selection

Once the various software deliverables have been identified, the criticality level shall be identified per the NR software quality criticality level process (SQCL) (ARP-68640-0305, 9/2/04). The software shall also have the JPL criticality level identified per the PSMP.

3 ROLES AND RESPONSIBILITIES

The PSMP calls out various roles that must be defined for the scope of the software development effort. These roles are identified here and are mapped into the NRPCT software development organization.

3.1 Descriptions

3.1.1 Software Manager

The software manager role as identified in the PSMP is performed by the manager of the KAPL Space I&C software group. The software manager is responsible for the delivery of software for the reactor I&C. Thus the software manager is responsible for requirements, design, implementation, testing at KAPL, and for coordinating with Bettis for the qualification testing.

3.1.2 System Engineer

The software system engineer role as identified in the PSMP is performed by the KAPL Space I&C group, and is cognizant of the overall I&C system. The system engineer is responsible for reviewing software requirements, software designs, and software implementations as part of the inspection process. This ensures that a proper level of system overview is present in the software development process.

3.1.3 Software System Engineer

The software system engineer role as identified in the PSMP is performed by the KAPL Space I&C Software group. The software system engineer role is responsible for defining the software requirements, and apportioning them between system components, and managing the interface between various software modules.

3.1.4 Software Architect

The software architect role as identified in the PSMP is performed by the KAPL Space I&C Software Lead. The software architect is responsible for defining the apportionment of software functionality between modules, and ensuring as much commonality as possible is maintained between the software versions.

3.1.5 Software Development Lead Engineer

The software development lead engineer role as identified in the PSMP is performed by the KAPL Space I&C Software developer assigned to specific modules. The software development lead engineer is responsible for defining the lower level design of individual software modules assigned and defining the unit test cases for these software modules.

3.1.6 Software Development Engineer

The software development engineer role as identified in the PSMP is performed by the KAPL Space I&C Software group. The software development engineer is responsible for helping to define the low level design of software modules, implementation of these software modules, and unit testing these software modules.

3.1.7 Software Test Engineer

The software test engineer role as identified in the PSMP is performed by the KAPL Space I&C Software group and the Bettis Space I&C Software test group. The KAPL software group is responsible for integration and test of the software modules. The Bettis software test group is responsible for final software system qualification testing.

3.1.8 Software Build Engineer

The software build engineer role as identified in the PSMP is performed by the KAPL Space I&C Software group. The software build engineer is responsible for the final integration and release of the software products.

3.1.9 Software Configuration Management Engineer

The software configuration management engineer role as identified in the PSMP is performed by the KAPL Space I&C Software group for the I&C software deliverables. The Bettis Space I&C Software test group will perform the configuration management functions for the independent Test Bed software models and tools.

3.1.10 Software Process Engineer

The software process engineer role as identified in the PSMP is performed by the KAPL Space I&C Software group. The software process engineer is responsible for developing these processes to maintain commonality with the Prometheus software development processes as much as is practicable.

3.1.11 Software System Administrator

The software system administrator role as identified in the PSMP is performed by the KAPL Space I&C Software group for the KAPL space I&C lab, also for the CTD in this lab. This role is performed by KAPL TIS for networked PC's.

The Software System Administrator role for the CTD development is performed by the Bettis Space I&C Software test group. This role is performed by Bettis network personnel for networked PC's.

3.1.12 Software Quality Assurance Engineer

The quality assurance engineer role as identified in the PSMP is performed by the KAPL QAE group for the Space I&C software deliverables. And will perform process compliance oversight and audits as appropriate. Bettis quality assurance will perform these functions for the Bettis CTD development effort.

3.1.13 Software Customer

The software customer role as identified in the PSMP is performed by NR. The software customer has ultimate approval over all software requirements and approval over final release of software products.

3.1.14 Software Line Organization

The software line organization role as defined in the PSMP is performed by the NRPCT. The software line organization performs the software development activities necessary to create high quality software deliverables for Prometheus Reactor I&C.

4 DEVELOPMENT PROCESSES

The detailed development processes are described in this section. This provides the inputs and outputs and specific process steps necessary to perform the tasks discussed in Section 3 for the development life cycle.

4.1 Process 1: Initial Software Requirements

ID: *SIC-1* / **Rev:** *0*	**Title:** *Initial Software Requirements*
Effective Date: *August 1, 2005*	**Supersedes:** *N/A*
Overview: Develop initial software functional requirements for Prometheus Reactor I&C	

Entry Criteria	Exit Criteria
• Preliminary or complete system FRD is available • Preliminary or complete system architecture is available • Requirements traceability process defined • Requirements management process defined • Requirements management tool available	• Requirements document Inspected • Requirements document design reviewed • Requirements submitted and approved by NR (Requirements include interface/telemetry requirements between various modules)

Inputs	Outputs
• System FRD (may be preliminary) • System architecture (may be preliminary) • Previous platform software FRDs if available • Functionality differences between platforms • Quality Attributes (Reliability, Fault Management, etc.) • Interface to Spacecraft Module/PCAD	• Inspection report • Design review report • Software FRD Submittal Letter • Approved software FRD • Approved requirements in Requirements Management Tool

Tasks

(a) Identify system FRD requirements allocated to software.

Use previous software FRDs as baseline if available

(b) Allocate software requirements between system tiers

(c) Refine definition of functional differences between platforms

(d) Define Interface Requirements, communication parameters

(e) Refine definition of Quality Attributes (Fault management, self-tests)

(f) Develop software requirements for target platform

(g) Place software requirements in requirements management tool

(h) Inspect software requirements

(i) Design review software requirements

(j) Submit software requirements to NR

(k) Incorporate NR comments and issue approved software FRD

Process Flow

See following Initial Software Requirements Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of comments from inspections and design reviews 3. Time to resolve comments 4. Number of comments from NR approval.	A. Local instruction of usage of Requirements Management tool B. Requirements Management Process C. Inspection Process D. Design Review Process

SIC-1 Initial Software Requirements Process Chart

4.2 Process 2: Initial Software Architecture

ID: *SIC-2* / **Rev:** *0*	**Title:** *Initial Software Architecture*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Develop initial software architecture for Prometheus Reactor I&C

Entry Criteria	**Exit Criteria**
• Preliminary or complete system FRD is available • Preliminary or complete system architecture is available • Initial or complete software FRD is available	• Software architecture inspected • Software architecture design reviewed • Software architecture documented with all relevant views • Software architecture provided to NR for information

Inputs	**Outputs**
• System FRD (may be preliminary) • System architecture (may be preliminary) • Functionality differences between platforms • Quality Attributes (Reliability, Fault Management, etc.) • Interface to other systems • Software architecture from previous platform • Software FRD (may be preliminary)	• Inspection report • Design review report • Documented software architecture

Tasks

a. Using software FRD, perform a Top-Down functional decomposition

If previous platform architecture is available, this should be the baseline for current architecture with minimal deviations

b. Develop structure for MOL and fault management using Quality Attributes

c. Use functional decomposition to modularize all software functions

d. Develop Data Flow Diagrams to show flow of data between modules

e. Develop modularity for as much similarity between platforms as possible (isolate platform specific functions to a module)

f. Document software architecture

g. Inspect software architecture

h. Design review software architecture

i. Provide software architecture to NR for information

Process Flow

See following Initial Software Architecture Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of comments from inspections and design reviews 3. Time to resolve comments 4. Number of comments from NR approval 5. Number of modules.	A. Local instruction on structured design architecture development B. Inspection Process C. Design Review Process

SIC-2 Initial Software Architecture Process Chart

4.3 Process 3: Risk Analysis

ID: *SIC-3* / **Rev:** *0*	**Title:** *Risk Analysis*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Perform risk analysis on Reactor I&C software requirements and architecture

Entry Criteria	**Exit Criteria**
• Preliminary or complete software FRD is available • Preliminary or complete software architecture is available	• Risks identified and documented • Risk mitigation strategies identified and documented • Risk impact and likelihood evaluated

Inputs	**Outputs**
• Software FRD (may be preliminary) • Software architecture (may be preliminary)	• Risks with evaluation and mitigation plans documented • SHA, SFTA, SFMECA

Tasks

a. Examine software requirements and software architectures to identify possible risks to cost/schedule due to complexity, uncertainty, or other criteria

b. Perform Software Hazard Analysis, Software Fault Tree analysis, and Software Fault Mode Effects and Criticality Analysis (SIC-A7)

c. Identify the severity of each risk, and the likelihood of each risk

d. Classify risks as highest for those likely to occur with severe impact, and low for those with low probability of occurrence and low impact

e. Develop mitigation plan for each risk

f. Document risks and mitigation plans

Process Flow

See following Risk Analysis Process Chart

Measures	References
1. Number and severity or risks 2. Time to complete task	A. Software Hazard Analysis, Fault Tree Analysis, and Failure Modes and Criticality Analysis process (SIC-A7)

SIC-3 Risk Analysis Process Chart

4.4 Process 4: Plan Incremental Builds

ID: *SIC-4* / **Rev:** *0*	**Title:** *Plan Incremental Builds*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Plan Incremental Builds for Reactor I&C Software for current platform

Entry Criteria	**Exit Criteria**
• Preliminary or complete software FRD is available • Preliminary or complete software architecture is available • Risk Analysis document completed	• One or more Incremental Build scheduled and resource loaded • Incremental Build Schedule sent to NR for information

Inputs	**Outputs**
• Software FRD (may be preliminary) • Software architecture (may be preliminary) • Risk Analysis document	• Integrated Build schedule resource loaded with all requirements allocated between builds

Tasks

a. Define number of Incremental Builds

b. Use architecture and risk analysis to allocate software functionality to the various builds

c. Define schedule to complete all modules (requirements through testing) for each Incremental Build

d. Resource Load schedule and ensure fit with overall project schedule and external program organizational needs

e. Provide Incremental Build Schedule to NR for information

Process Flow

See following Plan Incremental Builds Process Chart

Measures	References
1. Time to complete task	A. Integrated Master Schedule

SIC-4 Plan Incremental Builds Process Chart

4.5 Process 5: Develop and Release Incremental Build

ID: *SIC-5* / **Rev:** *0*	**Title:** *Develop and Release Incremental Builds*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Develop and Release Incremental Builds

Entry Criteria	**Exit Criteria**
• Preliminary or complete software FRD is available • Preliminary or complete software architecture is available • Risk Analysis document completed • Incremental Build Schedule Complete • Configuration Management Tool and Process available • Software Development Environment available • Test Environment available • Defect Tracking Tool available • Software under configuration control available (if not first iteration)	• Source Code and Executable Complete in Configuration Management • KAPL Software V&V complete • Release Readiness Review complete

Inputs	**Outputs**
• *Software FRD (may be preliminary)* • Software architecture (may be preliminary) • Risk Analysis document • Incremental Build Schedule	• *Source code and Executable released* • Inspection report issued • Unit Test report issued • Test Readiness Review Report issued • System Test report issued • Release Readiness Review Report issued

Tasks

a. Develop Detailed Requirements for Increment and Inspect (5.1)

b. Analyze and develop Detailed architecture for Increment and Inspect (5.2)

c. Design software Modules for Increment and Inspect (5.3)

d. Implement Module in source code, write unit test cases, Inspect code (5.4)

e. Unit Test Module implementation (5.5)

f. Integrate software modules on hardware (5.6)

g. Hold Test Readiness Review, perform system testing on SDVE (5.7)

h. Document and Release executable software, hold Release Readiness Review (5.8)

i. Bettis performs Independent Verification, Validation, and Testing (5.9)

j. Any defects encountered are entered into Defect Tracking database (5.10.a)

k. All defects are evaluated to determine proper disposition. Process repeated at earlier task as appropriate to correct defect. (5.10.b)

Process Flow

See following Develop and Release Incremental Build Process Chart

Measures	**References**
1. Time to complete task 2. Number of defects per stage 3. Time to correct defects 4. Code development productivity 5. Code inspection productivity	A. None

SIC-5 Develop and Release Incremental Build Process Chart

4.5.1 Process 5.1: Develop Detailed Requirements for Increment

ID: *SIC-5.1* / **Rev:** *0*	**Title:** *Develop Detailed Requirements for Increment*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Develop detailed functional requirements for functions contained in the current increment

Entry Criteria	**Exit Criteria**
• Preliminary or complete software FRD is available • Preliminary or complete software architecture is available • List of functions allocated to increment • Requirements Management Tool and Process available • Software Development Environment available • Defect Tracking Tool available	• Detailed Incremental Requirements Inspected • Requirements submitted to NR if changed from initial requirements • Requirements placed in Requirements Management tool

Inputs	**Outputs**
• System FRD (may be preliminary) • Software FRD (may be preliminary) • Software architecture (may be preliminary) • List of functions contained in increment • Incremental Build Schedule	• Updated software FRD • Inspection report issued • Design review report issued (if performed) • NR submittal letter issued

Tasks

a. Examine system FRD for new or changed information on software requirements contained in this increment

b. Determine if any refinement of software requirements is necessary and update

c. Ensure changes to software FRD are in Requirements Management tool

d. Inspect software FRD changes

e. Incorporated inspection findings

f. Design review may be performed if there is a large change from last design review

g. Incorporated design review findings if performed

h. Submit software FRD changes to NR

i. Incorporate NR comments into software FRD and issue

j. Baseline software FRD in Requirements Management tool

Process Flow

See following Develop Detailed Requirements for Increment Process Chart

Measures	References
1. Time to perform tasks 2. Number of comments from inspections and design reviews 3. Time to resolve comments 4. Number of comments from NR approval	A. None

SIC-5.1 Develop Detailed Requirements for Increment Process Chart

4.5.2 Process 5.2: Analyze/Develop Detailed Architecture for Increment

ID: *SIC-5.2* / **Rev:** *0*	**Title:** *Analyze/Develop Detailed Architecture for Increment*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Analyze/Develop detailed software architecture based on detailed functional requirements for functions contained in the current increment

Entry Criteria	**Exit Criteria**
• Preliminary or complete software FRD is available • Preliminary or complete software architecture is available • List of functions allocated to increment • Configuration Management Tool and Process available • Software Development Environment available • Defect Tracking Tool available	• Detailed software architecture Inspected • Revised architecture sent to NR for information

Inputs	**Outputs**
• System FRD (may be preliminary) • Software FRD (may be preliminary) • Software architecture (may be preliminary) • List of functions contained in increment	• Updated software architecture document • Inspection report issued • Design review report issued (if performed)

Tasks

a. Examine system FRD for new or changed information on software requirements contained in this increment

b. Determine if any refinement of software architecture is necessary and update (be mindful of effect on other platforms)

c. Document changes to software architecture

d. Inspect software architecture changes

e. Incorporate inspection findings

f. Design review may be performed if there is a large change from last design review

g. Incorporate design review findings if performed

h. Provide revised architecture to NR for information

Process Flow

See following Analyze/Develop Detailed Architecture for Increment for Increment Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of comments from inspections and design reviews 3. Time to resolve comments	A. None

SIC-5.2 Analyze/Develop Detailed Architecture for Increment Process Chart

4.5.3 Process 5.3: Detailed Module Design for Increment

ID: *SIC-5.3* / **Rev:** *0*	**Title:** *Detailed Module Design for Increment*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Develop detailed design for each module allocated to the current increment including data flow and internal structure.

Entry Criteria	**Exit Criteria**
• Preliminary or complete software FRD is available • Preliminary or complete software architecture is available • List of functions allocated to increment • Configuration Management Tool and Process available • Software Development Environment available • Defect Tracking Tool available	• Detailed module design inspected. • Module design documented

Inputs	**Outputs**
• System FRD (may be preliminary) • Software FRD (may be preliminary) • Software architecture (may be preliminary) • List of functions contained in increment	• Detailed Module design documented • Inspection report issued

Tasks

a. Determine detailed design of modules allocated to increment based on requirements, architecture, and any interface requirements.

b. Document module design

c. Inspect module design

d. Incorporate inspection findings

Process Flow

See following Detailed Module Design for Increment Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of comments from inspections 3. Time to resolve comments	A. None

SIC-5.3 Detailed Module Design for Increment Process Chart

4.5.4 Process 5.4: Module Implementation/Coding for Increment

ID: *SIC-5.4* / **Rev:** *0*	**Title:** *Module Implementation/Coding for Increment*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Implement modules for increment, write unit test cases, inspect source code, and desk check module implementation.

Entry Criteria	**Exit Criteria**
• Preliminary or complete software FRD is available • Preliminary or complete software architecture is available • List of functions allocated to increment • Detailed Module design • Configuration Management Tool and Process available • Software Development Environment available • Correction of defects found from prior testing • Defect Tracking Tool available	• Inspected Module implementation source code • Module Unit Test cases developed • Source Code in Configuration Management tool

Inputs	**Outputs**
• System FRD (may be preliminary) • Software FRD (may be preliminary) • Software architecture (may be preliminary) • Detailed Module design • List of functions contained in increment • Unit Test report (if correcting findings)	• Module source code • Code inspection report issued • Unit test cases developed

Tasks

a. Evaluate unit test findings (if any) and determine module changes necessary

b. Write or update source code for each module

c. Write unit test cases for complete MC/DC coverage of module

d. Inspect source code

e. Incorporate inspection findings, if inspection finding will not be incorporated at this time, it is to be entered into the Defect Tracking tool

f. Desk check module implementation (compile source code and run with stubs)

Process Flow

See following Module Implementation/Coding for Increment Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of comments from inspections 3. Time to resolve comments	A. Coding Standard

SIC-5.4 Module Implementation/Coding for Increment Process Chart

4.5.5 Process 5.5: Unit Test of Modules for Increment

ID: *SIC-5.5* / **Rev:** *0*	**Title:** *Unit Test of Modules for Increment*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Review and run unit test cases for modules, if defects found, correct defects via SIC-5.4

Entry Criteria	**Exit Criteria**
• Source Code for modules available • Unit Test cases for modules available • List of functions allocated to increment • Configuration Management Tool and Process available • Software Development Environment available • Test Environment available • Defect Tracking Tool and Process available • Software baselined and under configuration control	• All increment modules unit tested • Unit test cases reviewed • Source Code in Configuration Management tool • Test cases and results in configuration management

Inputs	**Outputs**
• Module Source Code • Module Unit Test Cases • Software architecture (may be preliminary) • Detailed Module design • List of functions contained in increment	• Reviewed Unit Test cases • Unit Test report issued

Tasks

a. Review unit test cases for completeness (reviewer must be independent of author)
b. Update unit test cases based on review
c. Perform unit testing, ensure complete MC/DC coverage
d. Archive results and unit tests in configuration management tool
e. Issue unit test report with findings
f. If defects are found, re-enter process SIC-5.4 to correct module implementation
   If defect can not be resolved at this time, enter into Defect Tracking database

Process Flow
See following Unit Test of Modules for Increment Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of findings from unit testing 3. Time to resolve comments	A. None

SIC-5.5 Unit Test of Modules for Increment Process Chart

4.5.6 Process 5.6: Module Integration for Increment

ID: *SIC-5.6* / **Rev:** *0*	**Title:** *Module Integration for Increment*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Integrate modules together and create executables. Run on hardware for preliminary checkout

Entry Criteria	**Exit Criteria**
• Source Code for modules available • Modules Unit Tested • Configuration Management Tool and Process available • Software Development Environment available • Test Environment available • Defect Tracking Tool and process available • Software baselined and under configuration control	• Executables placed in Configuration Management tool • Integration findings entered into Defect Tracking tool • All integration findings corrected

Inputs	**Outputs**
• Module Source Code • Target Platform	• Compiled executables under configuration management • Defects entered in tool and corrected

Tasks

a. Compile module source code and link together into executable

b. Enter compile or link errors into defect tracking tool

If any errors are encountered, develop disposition, and repeat previous process to correct errors before proceeding further

c. If compile/link is successful, download executable onto target platform and perform basic checkout

d. If any defects are detected during basic checkout, enter into defect tracking tool, develop disposition, and repeat previous processes to correct errors before proceeding further

e. If basic checkout is successful, place executables under configuration control

Process Flow

See following Module Integration for Increment Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of findings from integration 3. Time to resolve comments	A. None

SIC-5.6 Module Integration for Increment Process Chart

4.5.7 Process 5.7: System Testing for Increment

ID: *SIC-5.7* / **Rev:** *0*	**Title:** *System Testing for Increment*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Perform system testing of software executables for increment on SDVE

Entry Criteria	**Exit Criteria**
• System and software FRDs available • Software executables available • Configuration Management Tool and Process available • Software Development Environment available • Test Environment available • Defect Tracking Tool and Process available • Software baselined and under configuration control	• System Test Cases and results placed in configuration control • System Test findings entered into Defect Tracking tool • All System Test findings corrected

Inputs	**Outputs**
• Executable code • System/software FRDs • Target Platform	• Test Readiness Review report issued • System Test report issued • System Test Cases and results • Defects entered in tool and corrected

Tasks

a. Develop system test cases based on system and software FRDs
b. Review system test cases
c. Prepare Test Plan and hold Test Readiness Review
d. If review confirms test plan, platform, and executables are ready, perform system testing, otherwise correct
e. Archive Test Cases and results in Configuration Management tool
f. Place defects in defect tracking tool
g. Issue System Test report

Process Flow
See following System Testing for Increment Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of defects from Testing 3. Time to resolve defects	A. None

SIC-5.7 System Testing for Increment Process Chart

4.5.8 Process 5.8: Document and Release Incremental Build

ID: *SIC-5.8* / **Rev:** *0*	**Title:** *Document and Release Incremental Build*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Document and ensure all quality and configuration management tasks have been completed to release software executables for general use.

Entry Criteria	**Exit Criteria**
• Approved system and software FRDs available • Software source code and executables baselined and under configuration control • Code inspections, Unit Testing, and System Testing complete with reports issued • Defect reports reviewed and dispositioned • Software baselined and under configuration control	• Approval from Release Readiness Review through Configuration Control Board • Source Code, Executables, Configuration reports, and other documentation exported to Media for issuing to outside organizations • Release Readiness Review Report issued

Inputs	**Outputs**
• Executable code and source code • Approved system/software FRDs. • Inspection, Unit Test, and System Testing reports • Complete Traceability Matrix	• Configuration Documentation issued • Source Code and Executables issued • Release Readiness Review Report issued

Tasks

a. Ensure test program completed and test report issued.

b. Prepare release notes for source code and executables

c. Ensure source code and executables are properly baselined and documented per configuration management process

d. Review Traceability Matrix for complete coverage

e. Perform Release Readiness Review with NR and other stakeholders to ensure maturity of product for release (also review defects/open items)

f. Document review meeting in meeting minutes, address all open issues

g. Issue source code and executables with relevant documentation

Process Flow

See following Document and Release Incremental Build Process Chart

Measures	**References**
1. Time to perform tasks 2. Comments at Review	A. None

SIC-5.8 Document and Release Incremental Build Process Chart

4.5.9 Process 5.9: Independent Verification/Validation/Testing of Incremental Build

ID: *SIC-5.9* / **Rev:** *0*	**Title:** *Independent Verification/Validation/Testing of Incremental Build*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: An independent inspection and test program is performed at Bettis after release of an incremental build

Entry Criteria	**Exit Criteria**
• Approved system and software FRDs available • Software source code and executables baselined and under configuration control • Code inspections, Unit Testing, and System Testing complete with reports issued • Release Readiness Review completed	• Completion and documentation of all relevant Bettis testing and inspections • Documentation of any defects identified by Bettis

Inputs	**Outputs**
• Executable code and source code • Approved system/software FRDs	• Bettis inspection and test reports issued • Bettis Test Cases and results placed under configuration control

Tasks

a. Receive release materials for incremental build from KAPL
b. Prepare test cases from FRDs.
c. Perform software inspections based on NRPCT coding standard
d. Perform system testing based on FRD functionality
e. Document any inspection or system test findings
f. Issue test and inspection reports

Process Flow

See following Independent Verification/Validation/Testing of Incremental Build Process Chart

Measures	**References**
1. Time to perform tasks 2. Number of defects from testing 3. Number of findings from inspection	A. None

SIC-5.9 Independent Verification/Validation/Testing for Incremental Build Process Chart

4.5.10 Process 5.10a: Defect/Change Identification for Incremental Build

ID: *SIC-5.10a* / **Rev:** *0*	**Title:** *Defect/Change Identification for Incremental Build*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: For Integration, System Testing, or Independent V&V, defects are identified and placed into defect tracking tool.

Entry Criteria	**Exit Criteria**
• Defect Tracking Tool available • Defect found in project step	• Finding discussed with cognizant engineer and placed in Defect Tracking Tool

Inputs	**Outputs**
• Defect identified in source code or executable	• Defect Documented in Defect Tracking Tool

Tasks

a. Defect is identified during inspection, Unit Test, Integration, System Test, or Independent V&V; if applicable, tester should attempt to replicate defect as this will help to narrow down possible defect source

b. Defect is discussed with module cognizant engineer to ensure defect is properly characterized

c. Defect is entered into Defect Tracking Tool with at least:

Date of defect, brief description, affected module, test script, test case, inputs, expected and actual outputs, detailed description of defect

d. Cognizant engineer shall concur to defect description in tool

e. Notify stakeholders of defect

Process Flow

See following Defect Identification for Incremental Build Process Chart

Measures	References
1. Number of defects 2. Time to enter defect and obtain cognizant engineers concurrence	A. None

SIC-5.10a Defect Identification for Incremental Build Process Chart

4.5.11 Process 5.10b: Defect/Change Disposition for Incremental Build

ID: *SIC-5.10b* / **Rev:** *0*	**Title:** *Defect/Change Disposition for Incremental Build*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Any defects that have been identified must be evaluated and proper corrective action assigned and performed.

Entry Criteria	**Exit Criteria**
• Defect Tracking Tool available • Undisposed defects identified and entered into the tool	• Corrective action identified, implemented, and verified

Inputs	**Outputs**
• Undisposed identified defects	• Defects have corrective action implemented and verified • Defect closed out in tool

Tasks

a. Each defect identified in tool is evaluated by Configuration Control Board, including tester and developer

b. Once source of defect is identified, it is documented in tool

c. Possible corrective actions are identified in tool

Corrective actions may include reimplementation, design change, or requirements changes

d. Corrective action authorized by CCB, developer performs corrective action

CCB scope is determined by scope of corrective action. For a simple local implementation change, a smaller scope CCB may be required, but for an architecture or requirements change, NR may have to be involved as well

e. Once software has been successfully retested, defect is closed out in tool

Process Flow

See following Defect Disposition for Incremental Build Process Chart

Measures	References
1. Number of defects 2. Time to provide solution for defect and successfully retest	A. None

SIC-5.10b Defect Disposition for Incremental Build Process Chart

5 ANCILLARY PROCESSES

5.1 Process A1: Configuration Management

ID: *SIC-A1* / **Rev:** *0*	**Title:** *Configuration Management*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Requirements, design, source code, executables, test cases, and results require configuration control and change management

Entry Criteria	**Exit Criteria**
• Configuration Management tool available • Requirements Management tool available • Document archive tool available	• Article place in configuration control, change history updated, article version appropriately identified, and collection of configuration items properly baselined • Changes approved by CCB, changes verified once made

Inputs	**Outputs**
• Article created that must be placed in CM system • Revision history and number identified	• All articles "checked in" with proper labeling and revision history • CCB meeting minutes issued for authorized changes to configuration items

Tasks

a. Type of article identified to determine configuration management system to be used:
   1. Requirements – requirements management system
   2. Design (architecture) – issued as letter and placed in document archive tool (ADSARS)
   3. Source code, executables – Configuration management system
   4. Test cases, test results – Configuration management system
   5. Inspection, design review, and Test reports – issued as letter and placed in document archive tool (ADSARS)

b. Requirements are placed in Requirements Management tool:
   1. User rights: Author has full change access, other users only have read access, administrator has full access to set user access
   2. On creation, requirements placed in tool and edited in tool
   3. Requirements are inspected and changes placed in tool

4. Requirements are design reviewed and changes placed in tool
5. Requirements are baselined as "Proposed" and a revision number is assigned (start at 0, e.g. "Proposed Revision 0.0")

   Revision has major and minor number X.Y, major incremented for large changes, minor incremented for smaller changes (as determined by CCB)
6. Requirements are exported from the tool into a document for sending to NR for approval, submittal letter is placed in document archive tool (ADSARS).
7. Once NR approval has been received, and NR comments have been incorporated into the tool, requirements are rebaselined, and the "Proposed" designation is removed
8. Approved requirements are exported from tool into a document and issued for information, issuing letter is placed in document archive tool (ADSARS)
9. Any requirements change identified from Defect Disposition or other sources must be evaluated by CCB, and if accepted, changes must repeat steps b.3 through b.8.

c. Software design and architecture:

1. Software design and architecture must be inspected and design reviewed
2. Software design and architecture are captured in a document
3. Document is issued via letter and placed in document archive tool (ADSARS)
4. Any design or architecture change identified from Defect Disposition or other sources must be evaluated by the CCB, and if accepted, change shall be inspected (and design reviewed if large in scope), then reissued via letter and placed in the document archive tool (ADSARS)

d. Source Code and Executables

1. User rights: Author has full change access, other users only have read access, administrator has full access to set user access
2. Each source code file or executable is created and placed in the Configuration Management tool
3. Files are checked in and out by the author for implementation
4. Once implementation is complete, the files are baselined as "Revision 0.0" (X.Y where X is the branch (thread) and Y is the file revision, the first (and main) branch is 0, but it is envisioned that the files will be shared across platforms, and may need platform specific changes, thus necessitating a new branch to track the same file in multiple branches) and code inspected
5. Code inspection comments are incorporated into the files which are then rebaselined with revision history (date, author, brief description of changes) added and the revision number is incremented
6. Once unit testing is completed, findings are incorporated, and the files are rebaselined with updated revision history and revision numbers, an application version number is applied to all files that make up an executable (e.g. Version 1.0) At this point, CCB approval is necessary to further change files

7. Once integration is performed any integration findings must be approved by the CCB to make changes to the source files. Any files to be changed must have revision history and revision numbers updated, and once all integration changes are complete, the collection is rebaselined and a new version number is assigned (e.g. Version 1.1). The generated executable is then updated with a new version number.
8. System testing is performed on the compiled executable, and any changes identified by system testing must be approved by the CCB, then the product must have source file revision history and revision numbers updated, the collection must be rebaselined with the application version number updated (along with change history for the application), the source files proceed through integration to create a new executable, and this executable is labeled with an updated version number.
9. Later increments repeat steps 2 through 8

e. Test cases and results:

1. Unit test cases and system test cases are placed in the configuration management tool once generated
2. Once the test cases have been reviewed, the cases are baselined as "Revision 0.0" (X.Y where X is the branch (thread) and Y is the file revision, the first (and main) branch is 0, but it is envisioned that the files will be shared across platforms, and may need platform specific changes, thus necessitating a new branch to track the same file in multiple branches).
3. Once the test cases have been run, the results are archived in the configuration management tool until needed for issuing the test report

f. Inspection reports, design review reports, test result reports:

1. The various reports are prepared in letter format and placed through the standard letter review process (primary design check, management review, administrative review, issue)
2. Once the letter is issued, it is archived in the document archive tool (ADSARS)

CCB – Configuration Control Board: Used to review potential changes to items under configuration control, and then accept or reject change requests

1. The CCB consists of stakeholders relevant to the scope of changes being requested. Members consist of : NR, Manager Space I&C Systems & Software (Chairman of CCB), Cognizant software developer (requesting change), System Cognizant engineers, Test Engineers, Bettis Space I&C Engineers (when change may impact CTD software or delivery dates of releases to Bettis), BPMI Engineers (when change may effect program vendors of final delivery of Incremental Build), Other Prometheus program organizations as appropriate
2. Change request must be submitted to CCB prior to meeting. Change request must contain affected modules, scope of change, estimated scope of retest, estimated schedule for completion of change and follow-up qualification, and defects being addressed
3. CCB meeting must be scheduled, and items are to be reviewed at the meeting, a

disposition for each requested change is determined, and meeting minutes are issued

Process Flow
See following Configuration Management Process Chart

Measures	**References**
1. None	A. None

SIC-A1 Configuration Management Process Chart

5.2 Process A2: Requirements Traceability

ID: *SIC-A2* / **Rev:** *0*	**Title:** *Requirements Traceability*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Clear traceability must be established between the requirements, implementation, and test cases. This traceability must be bidirectional to be able to verify complete implementation and coverage.

Entry Criteria	**Exit Criteria**
• Configuration Management tool available • Requirements Management tool available	• Traceability matrix established for bidirectional traceability between all artifacts

Inputs	**Outputs**
• Approved software and system FRD • Software architecture and design Documents • Source Code • Unit and system test cases	• Bidirectional traceability matrix between source code and requirements (and design) • Bidirectional traceability matrix between requirements, test cases, and results.

Tasks

a. Once FRD has been established and software architecture/design is created, a traceability matrix shall be created to identify which requirements are satisfied in each module, and which modules satisfy each requirement (bi-directional)

b. Once source code is created, a bi-directional traceability matrix shall be created between the requirements and the code.

c. Once Unit and system test cases have been created a bidirectional traceability matrix shall be created between the test cases and the source code.

d. Any time one of the inputs is updated, the traceability matrix shall be examined to determine if updates are necessary, and then updated.

e. The traceability matrix shall be provided as part of the release documentation for a build

Requirement Management tool may support linking for traceability matrix

Process Flow

See following Requirements Traceability Process Chart

Measures	References
1. None	A. None

SIC-A2 Requirements Traceability Process Chart

5.3 Process A3: Inspections

ID: *SIC-A3* / **Rev:** *0*	**Title:** *Inspections*

Effective Date: *August 1, 2005*	**Supersedes:** *N/A*

Overview: Inspections are performed on work products to catch defects early and help improve the product.

Entry Criteria	**Exit Criteria**
• Configuration Management tool available • Requirements Management tool available • Applicable standards available • Applicable checklists available • Inspection work product baselined and under configuration control	• Inspection report with comments issued

Inputs	**Outputs**
• Software and system FRD • Software architecture and design documents • Source code • Unit and system test cases	• Inspection report issued • Findings placed in defect tracking tool if appropriate

Tasks

a. An inspection group, independent of work product author, is established of nominally 3 to 5 engineers, a chairman/scribe, and the author (to present). The members may be NRPCT or from other Prometheus project organizations (assuming clearance and NTK can be established).

b. Requirements, architecture, design, code, and test case inspections:

1. Kickoff meeting held with author and all inspectors, author presents requirements and walks inspectors through document
2. Requirements are baselined, and distributed 3 weeks prior to the review meeting to all members along with any applicable standards and checklists
3. Inspectors review requirements and provide comments to chairperson 1 week prior to review meeting, chairman provides comments to author for preliminary disposition
4. Review meeting walks through inspector comments and author preliminary responses, and any new comments are documented at meeting
5. All comments are provided to author via formal meeting minutes, author then

formally replies with response/closeout of comments. Alternately, comments could be discussed prior to issue of meeting minutes and meeting minutes could document both comments and resolutions

6. Any unresolved comments are placed in Defect Tracking tool for later disposition

c. Inspection report shall provide summary information related to number and types of comments

Process Flow

See following Inspection Process Chart

Measures	**References**
1. Number of comments 2. Time to resolve comments	A. Requirements standard B. Design standard C. Coding standard D. IEEE Std. 1028

SIC-A3 Inspection Process Chart

5.4 Process A4: Design Reviews

ID: *SIC-A4* / **Rev:** *0*	**Title:** *Design Reviews*

Effective Date: *August 1, 2005*	**Supercedes:** *N/A*

Overview: Formal review performed on requirements and software architecture per KQA-10.

Entry Criteria	**Exit Criteria**
• Configuration Management tool available • Requirements Management tool available • Applicable standards available • Applicable checklists available	• Design review report issued • Design review closeout issued

Inputs	**Outputs**
• Software and system FRD • Software architecture and design	• Design review report issued • Design review closeout issued • Findings placed in defect tracking tool if appropriate

Tasks

a. A design review group, independent of work product author, is established of nominally 5 to 8 engineers, a chairman/scribe, and the author (to present). The members may be NRPCT or from other Prometheus project organizations (assuming clearance and NTK can be established). Members are chosen to be outside of Space I&C group with independence but expertise in the area of review

b. Requirements, architecture, design review (per KQA-10):

   1. Design review Chairperson selected and a design review is formally requested with a design review number assigned by SQA
   2. Kickoff meeting held with author and all reviewers, author presents requirements or architecture and walks reviewers through document
   3. Requirements or architecture is baselined, and distributed 3 weeks prior to the review meeting to all members along with any applicable standards and checklists
   4. Reviewers review requirements or architecture and provide comments to chairperson one week prior to review meeting, chairman provides comments to author for preliminary disposition
   5. Review meeting walks through design review comments and author preliminary responses, and any new comments are documented at meeting.

6. All comments are categorized as either Findings (which require formal response) or Observations (which do not require formal response)
7. All comments are provided to author via formal meeting minutes, author then formally replies with response/closeout of comments. Alternately, comments could be discussed prior to issue of meeting minutes and meeting minutes could document both comments and resolutions.
8. Any unresolved comments are placed in Defect Tracking tool for later disposition.

c. Design review report shall provide summary information related to number and types of comments (findings and observations).

Process Flow

See following Inspection Process Chart

Measures	**References**
1. Number of comments	A. Requirements standard
2. Time to resolve comments	B. Design standard
	C. Coding standard
	D. KQA-10

SIC-A4 Design Review Process Chart

5.5 Process A5: Test Readiness Reviews

ID: *SIC-A5* / **Rev:** *0*	**Title:** *Test Readiness Reviews*

Effective Date: *August 1, 2005*	**Supercedes:** *N/A*

Overview: Review to ensure test platform, software, test cases, and the test plan are ready to commence a test program.

Entry Criteria	**Exit Criteria**
• Test plan available • Executables ready for testing • Test environment ready	• Test Readiness Review meeting minutes issued • All review comments and concerns have been addressed

Inputs	**Outputs**
• Test plan • Test environment • Test cases • Executable software	• Test Readiness Review minutes issued

Tasks

a. Ensure test plan is complete, test bed configuration is documented, executable configuration is documented, and test cases are reviewed

b. Schedule Test Readiness Review; include Tester, Cognizant developer, Manager Space I&C Systems and Software, Test director, System cognizant engineer, and others as appropriate

c. At Readiness Review, test plan shall be walked through, and there should be confirmation of configuration of Test environment, Test cases, and executable software

d. Any comments identified at the Readiness Review shall be documented in the meeting minutes and resolved before testing can commence

e. Test Readiness Review meeting minutes are issued

Process Flow

See following Test Readiness Review Process Chart

Measures	References
1. Number of comments 2. Time to resolve comments	A. Test Plan template

SIC-A5 Test Readiness Review Process Chart

5.6 Process A6: Release Readiness Reviews

ID: *SIC-A6* / **Rev:** *0*	**Title:** *Release Readiness Reviews*

Effective Date: *August 1, 2005*	**Supercedes:** *N/A*

Overview: Review to ensure executable has completed all necessary quality steps for general release.

Entry Criteria	**Exit Criteria**
• Executable successfully completed all inspections, unit testing, and system testing • Executable configuration documentation is complete • All defects have been satisfactorily disposed and closed out	• Release Readiness Review Report issued • Source code, executables, and all associated documentation are available for general use

Inputs	**Outputs**
• Code inspection report • Approved software FRD • Unit Test report • System Test report • Release Notes/ Change List	• Release Readiness Review Report issued • Source Code/Executables/ Documentation available for general use

Tasks

a. Ensure necessary inputs are all available

b. Schedule Release Readiness Review, include CCB members, including NR, the Cognizant developer, System cognizant engineer, Tester, Manager Space I&C Systems and Software, Test Director, Bettis Space I&C personnel, and representatives from other Prometheus organizations as appropriate

c. At Release Readiness Review, the configuration documentation for the source code and executables shall be examined, inspection report, Unit Test report, and System Test report should have been issued, software FRD shall have been approved and issued, and all defects shall have been dispositioned and closed.

d. The Release Readiness Review Report shall be issued

e. If accepted by the CCB, the source code and executables are made available for general use

Process Flow

See following Release Readiness Review Process Chart

Measures	**References**
1. Number of comments 2. Time to resolve comments	A. None

SIC-A6 Release Readiness Review Process Chart

5.7 Process A7: Software Hazard Analysis, SFTA, and SFMECA

ID: *SIC-A7* / **Rev:** *0*	**Title:** *Software Hazard Analysis, SFTA, and SFMECA*

Effective Date: *August 1, 2005*	**Supercedes:** *N/A*

Overview: Review of requirements and design to ensure faults and hazards are understood and mitigated to the maximum extent practical

Entry Criteria	**Exit Criteria**
• Software FRD available • Software architecture available	• Report prepared on Software Hazard Analysis, Fault Tree Analysis, and Failure Mode Effects and Criticality Analysis

Inputs	**Outputs**
• Software FRD • System FRD • Software architecture	• Software Hazard Analysis • Software Fault Tree Analysis • Software Failure Modes, Effects and Criticality Analysis

Tasks

f. Perform Software Hazard Analysis:

1. [RESERVED]

g. Perform Software Fault Tree Analysis:

1. [RESERVED]

h. Perform Software Failure Modes, Effects and Criticality Analysis:

1. [RESERVED]

Process Flow

See following Software Hazard Analysis, SFTA, and SFMECA Process Chart

Measures	References
1. Number of comments 2. Time to resolve comments	A. None

SIC-A7 Software Hazard Analysis, SFTA, and SFMECA Process Chart

5.8 Process A8: Auditing and Self-Assessments

ID: *SIC-A8* / **Rev:** *0*	**Title:** *Auditing and Self-Assessments*

Effective Date: *August 1, 2005*	**Supercedes:** *N/A*

Overview: Performance of auditing and self-assessments to ensure compliance to processes.

Entry Criteria	**Exit Criteria**
• Periodic self-assessment period • SQA audit • Spot check	• Process audited, results documented, and any defects corrected along with any training or process improvement needs addressed

Inputs	**Outputs**
• Software Development Plan • Work Products	• Audit or internal review report • Audit or internal review response

Tasks

a. Process or work product to be audited is chosen

b. Audit performed to ensure compliance to written process, or to ensure work product satisfies intended function (e.g. audit of defect resolution)

c. Once audit or self-assessment is complete, results are documented and issued

d. Response is generated to perform corrective actions including retraining, or process improvement

Process Flow

See following Auditing and Self-Assessments Process Chart

Measures	**References**
1. Number of comments 2. Time to resolve comments	A. None

SIC-A8 Auditing and Self-Assessments Process Chart

6 OTHER PROCESSES [RESERVED]

Processes identified here are identified in the PSMP and will be provided as ancillary processes when further definition has been completed.

Software Classification
The PMSP defines classification levels for software to identify which software requires greater verification and validation based on the criticality of the software (consequences of software failure). This will be performed using the NNPP SQCL.

Risk Management
Risk management includes the identification of risks, the determination of mitigation strategies for risk, and the active review and application of those strategies throughout the life of the product.

Process Modification
As processes and used throughout the life of the project, improvements will be identified and have to be incorporated into the process documents.

Supplier Agreement Management
N/A

Metrics
Measures of performance used to help gauge project status and improve future planning.
Project Status, Risk, Defects, Earned Value, Schedule, Cost, Staffing, Functionality, Requirements, & others per PSMP.

Acceptance and Deployment
Process of certifying testing and releasing the software.

Operations and Maintenance
Process for maintaining software once released.

Corrective Actions
Tasks performed in response to problem reports.

Lessons Learned
Collecting information on unplanned events that can be applied to improve the process in the future.

Process Improvements
See process modification.

Acquisition
Process used to procure hardware and software development tools.

Causal Analysis and Resolution
Determining the root cause of a problem and fixing the both the problem and the process.

NRPCT Reactor Module Software Development Plan (NRPCT-RM-SDP-001)

NRPCT I&C Software Team

July 2005

This page intentionally left blank.

Revision History

Revision	Author	Date	Change Synopsis	Reason for Change
Draft	T. Hamilton D. Schroeder	6/30/05	Original	

Table of Contents

1 INTRODUCTION

1.1 Identification

This is the NRPCT Reactor Module Software Development Plan (SDP) for the Prometheus project.

1.2 Purpose

The SDP establishes the mission specific management, development, verification and validation processes for the reactor module I&C software of project Prometheus. Each mission has its own specific SDP detailing how the processes specified in the NRPCT Reactor I&C Software Development Process Manual (SDPM) are to be implemented for that mission. The mission for the Reactor Module SDP is based on a Nuclear Electric Propulsion (NEP) spacecraft. This work includes but is not limited to the flight software, ground telemetry and analysis software, test beds, and vendor developed sensor and actuator interface software. These practices ensure that the reactor module I&C software is of sufficient quality to meet the Prometheus needs, particularly the paramount need of safety in the spacecraft. This document implements the processes identified in the NRPCT Reactor I&C Software Development Process Manual (SDPM).

1.3 Basis and Standards

This document expands upon and traces to the Prometheus Software Management Plan (PSMP), JPL document #982-00046. The PSMP defines the high level software requirements and processes to be used throughout project Prometheus. Although the Memorandum of Understanding and the Memorandum of Agreement between NASA and Naval Reactors does not give JPL approval authority over Reactor Module software development, the NRPCT desires to maintain as much commonality in software processes as possible. Following the principles established in the PSMP helps to maintain commonality among software developed for the Reactor Module, Spaceship Module, Mission Module, and Ground System software.

This document incorporates guidance from the Prometheus Software Quality Assurance Requirements (JPL Document #982-00038) to ensure that there is commonality with software qualification across the Prometheus project.

This document incorporates guidance from several NRPCT standards and policies. The SDP incorporates guidance from the NR Software Engineering Policy (as documented in Bettis Letter No. B-REO(M)CD-008, 3/16/05). The SDP will determine the software quality criticality level (SQCL) for the flight and ground software per the NNPP Standard for Software Qualification by Criticality Level (as issued for three prime concurrence by KAPL Letter No. ARP-68640-0305, 9/2/04).

1.4 Division of Responsibilities

The Prometheus project has several organizations involved in software development; these include JPL (Mission Module and Ground Data System), the spacecraft contractor (Spacecraft Module), and the NRPCT (Reactor Module). The spacecraft contractor has further split the SM work between themselves (Control and Data Handling) and Hamilton

Sundstrand (PCAD). The NRPCT has split the Reactor Module software development work between:

1 KAPL - RM Flight and Ground System software development and integration)
2 Bettis - (CTD development and Independent Verification and Validation of RM Flight and Ground software, also sensor interface software development (requirements and verification, implementation by vendors)),
3 BPMI - (contract out sensor development work to vendors for fabrication, and sensor software development at vendors).

In the above role, KAPL develops and integrates the RM flight and ground software, Bettis performs independent V&V of the RM flight and ground software. Bettis also develops the CTDs for both its independent V&V, and for delivery to KAPL as part of the KAPL integration and test process. BPMI takes part in various software inspections and design reviews, and also is the contracting agency responsible for handling vendor development of sensors and sensor interface card hardware and software. Bettis will also perform independent V&V upon vendor developed sensor software. Once incremental builds are developed and released, they are handed off to JPL and the spacecraft contractor for integration within the Space Vehicle Test Beds for testing with the other developed systems. Models for the C&DH Flight Computer Assembly and the PCAD will be provided by the spacecraft contractor for integration with the KAPL and Bettis CTDs to allow composite testing with the RM I&C hardware and software.

1.5 Notation & Terminology

The NRPCT SDP follows the same Notation & Terminology of the PSMP.

1.6 Document Hierarchy

As with any project, there are many documents that govern the flow of work performed for that project. A document hierarchy provides a roadmap to aid in understanding of the layout and structure for documents important to Prometheus software development. The highest level document used for Reactor Module software development is the Project Software Management Plan (PSMP), which lays out process requirements that the NRPCT has agreed to work with to achieve more commonality with the other Prometheus software development organizations. Under the PSMP is the NRPCT Reactor I&C Software Development Process Manual (SDPM). This document provides the NRPCT specific implementation and customization of the guidance from the PSMP for processes based on an incremental software life cycle. The implementation of these processes for a specific mission is then handled in the Software Development Plan

The RM SDP (NRPCT-RM-SDP-001) provides mission specific definition for the software lifecycle, methodology, and implementation language for the RM flight software. The SDP also provides definition for the Roles and Work Products defined by the PSMP and the SDPM. The appendices for the SDP provide traceability to the PSMP, SQCL, NNPP SEP, and other influencing documents.

A series of documents provide further definition for items identified in the PSMP and are subordinate to the SDP. These documents include:

1 The KAPL RM Flight Software Development Plan (KAPL-RM-FSW-SDP-001) which defines the resources and specific tasks for the RM flight software.
   - KAPL Flight Software Work Breakdown Structure (KAPL-RM-FSW-WBS-001), which defines the tasks necessary to complete software development for all of the deliverables used for the Prometheus RM Flight Software.
   - The KAPL Flight Software Schedule (KAPL-RM-FSW-SCHD-001), which provides the schedule for all of the items identified in the WBS.

2 The KAPL RM Ground Software Development Plan (KAPL-RM-GSW-SDP-001) which defines the resources and specific tasks for the RM ground software.
   - KAPL Ground Software Work Breakdown Structure (KAPL-RM-GSW-WBS-001), which defines the tasks necessary to complete software development for all of the deliverables used for the Prometheus RM Ground Software.
   - The KAPL Ground Software Schedule (KAPL-RM-GSW-SCHD-001), which provides the schedule for all of the items identified in the WBS.

3 The Bettis Test Bed Development Plan (BETTIS-RM-TB-DP-001), which *provides CTD specific development tasks, layout, and goals.*
   - The Bettis RM Test Bed Work Breakdown Structure (BETTIS-RM-TB-WBS-001), which defines all of the tasks necessary for CTD development, vendor sensor card development, and independent V&V of the RM software.
   - The Bettis RM Software Schedule (BETTIS-RM-TB-SCHD-001), which defines the schedule for all of the items identified in the WBS.

4 The Vendor Sensor Interface Software Development Plan [RESERVED]

1.7 Definitions

N/A

1.8 Acronyms

Table 1

Acronym	Definition
CL	Criticality Level
CTD	Composite Test Device
DSS	Deep Space System
I&C	Instrumentation and Control
FRD	Functional Requirements Document
IEEE	Institute of Electrical and Electronics Engineers
JPL	Jet Propulsion Laboratory
MM	Mission Module
NRPCT	Naval Reactors Prime Contractor Team
PCAD	Power Conditioning and Distribution
PSMP	Prometheus Project Software Management Plan
PSR	Project Software Requirements
RM	Reactor Module
SDVP	Software Development and Verification Platform
SFTA	Software Fault Tree Analysis

Acronym	Definition
SFMECA	Software Failure Modes, Effects and Criticality Analysis
SHA	Software Hazard Analysis
SIC	Space I&C
SM	Spacecraft Module
SQCL	Software Quality Criticality Level
V&V	Verification and Validation

1.9 References

N/A

2 OVERVIEW

2.1 Description

The Prometheus Deep Space System (DSS) is composed of the Reactor Module and the Spacecraft Module (SM). The DSS is the reusable portion of the Prometheus project; subsequent missions will tailor the DSS, develop a mission specific set of science instruments and deliver the science instruments as part of a Mission Module (MM). These Reactor and Spacecraft modules, as well as the Mission Module, are developed by different organizations and the baseline assumption is that there will be software associated with all of these modules. NRPCT has responsibility for the Reactor Module.

The Prometheus Reactor Module (RM) includes the nuclear reactor, reactor instrumentation and control, reentry shield and radiation shielding. The reactor control and the Spacecraft Module power conversion segment are tightly coupled and thus close coordination between the Reactor and Spacecraft Module teams is required. The Reactor Module Instrumentation and Control team produces the software and hardware required to control the reactor.

The Reactor Module notional I&C architecture utilizes a two tiered system design. The top layer is the supervisory system, which contains one "hot" supervisor plus "warm" and "cold" backup supervisors. The supervisor is responsible for communicating with the spacecraft computer (in the SM), both to accept commands, and to relay telemetry. The hot supervisor is determined through a hardware arbitrator system also known as the Fault Management Assembly (FMA). The supervisor then communicates with the controller tier in the I&C system. There are four reactor controllers, each one responsible for monitoring plant sensors, and performing control and protective functions through reactivity control. The reactor controllers also receive feedback from and send commands to the PCAD system to react to changes in the power conversion and heat rejection systems. Output from the reactor controllers is directed through a coincidence system prior to commanding a change in the reactivity control devices. Both the supervisor and the controllers contain software. There is also a Ground system that displays data from the Reactor Module as relayed through the Spacecraft computer. The system is displayed in Figure 1.

The NRPCT is developing the software for the Reactor Supervisor, the Reactor Controller, and the portion of the Ground System that is responsible for communicating with the Reactor Module. NRPCT is responsible for defining the requirements for the

sensor interface software; sub tier vendors will implement the sensor interface software, and NRPCT will qualify the sensor interface software. Other groups are responsible for developing the Spacecraft Module software (spacecraft contractor), the PCAD software (Hamilton Sundstrand), and the Mission Module software (JPL). The NRPCT will also be developing test fixtures and software models for simulating the reactor dynamics, and emulating sensors and I&C channels. These collectively are known as the Composite Test Device (CTD) software.

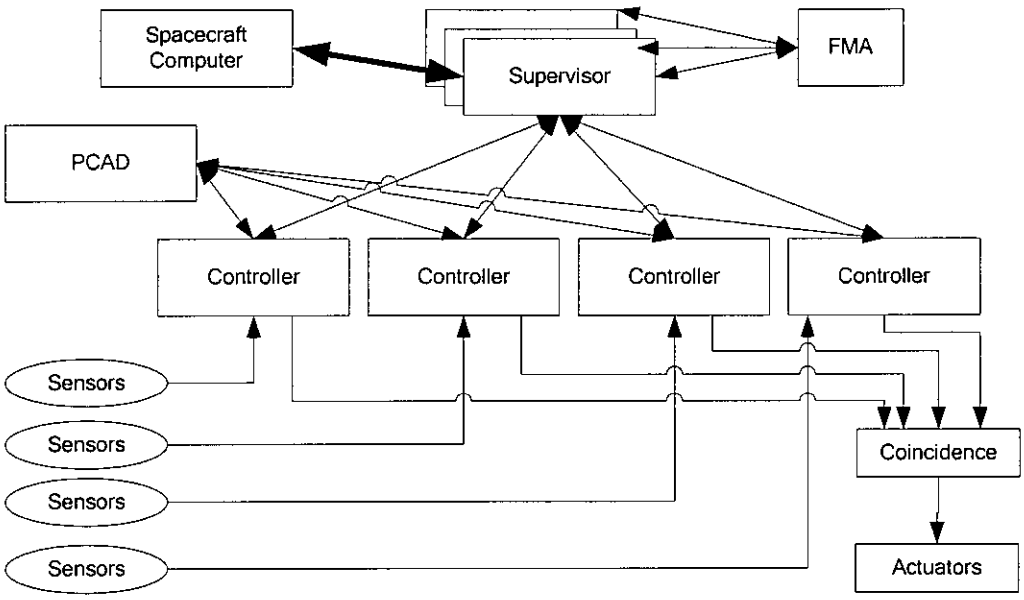


Figure 1: System Architecture

2.2 Project Phases

The overall system for Prometheus must be tested before the spaceship is launched. This testing is performed in several phases, with I&C systems developed to meet the needs of each test phase. The following development phases have been designated for the Reactor Module that corresponds to the applicable test phases:

1) Engineering Model (EM): This consists of an engineering model of the I&C System linked together with the PCAD and SM Flight computer systems. This is then connected to a non-nuclear test loop with heaters to simulate the reactor heat. This enables an end-to-end testing of the various interfaces along with input from actual system sensors. This will most likely consist of a single channel of supervisor and a single controller.
2) Qualification Model (QM): This consists of an end-to-end test of the various DSS module interfaces with the most prototypical representation of the flight unit

reactor. This will also be a non-nuclear test with heaters simulating the reactor. Commands will be issued through a simulated link with ground.

3) Ground Test Reactor 1 (GTR1): This consists of a ground based prototype reactor connected with a Brayton unit. The I&C for this prototype will not be identical to the flight software since GTR1 must accommodate many up and down power maneuvers, manual operation, and be capable of full shutdown. There will most likely be some form of an operator interface panel for manual control of the reactor.
4) Ground Test Reactor 2 (GTR2): This consists of a second ground based prototype much closer to the flight unit. This will require the option of manual control and full shutdown, but will have sensors and control algorithms very close to the final form for flight. There also may be an interface between the Ground System software and the Reactor Module.
5) Prometheus 1 (P1): This consists of the first Prometheus spacecraft, and will most likely be an inner solar system mission, possibly to the Moon or the asteroid belt. This would consist of the final flight software.
6) Prometheus 2 (P2): This consists of the second Prometheus spacecraft, and will most likely be an outer solar system mission, perhaps to Jupiter. Ideally, the software will be identical between P1 and P2, but if necessary, any changes required due to further test experience or experience gained in P1 would be incorporated into the P2 software.

The NRPCT is responsible for the products listed below in Table 2. Each of the products for EM, QM, GTR1, GTR2, P1, and P2 consist of the Reactor Supervisory computer software, the Reactor Controller computer software, sensor interface software, and the Reactor Ground Mission/Operations software.

Table 2: Software Deliverables and Classification

System	Module	SDE	Product Type	PSMP Baseline Product Class	NNPP Software Quality Criticality Level (SQCL) <preliminary>
DSS	Reactor	NRPCT	RM Flight Software for EM		
		NRPCT	RM Flight Software for QM		
		NRPCT	RM Flight Software for GTR1	B	A
		NRPCT	RM Flight software for GTR2	B	A
		NRPCT	RM Flight Software for P1	B	A
		NRPCT	RM Flight Software for P2	B	A
		NRPCT	RM Test Bed Software	C	TBD

The PMSP Product class has level 'B' as mission critical, level 'C' as mission support, and level 'A' is human rated.
The SQCL has Criticality Level 'A' as safety critical.

Table 3 Deliverables Matrix

SDPM Process Step	Process Deliverable	Flight Software	Ground Software	Test Bed
4.1: SIC-1 – Initial Requirements	Inspection Report	X	X	
	Design Review Report	X	X	
	FRD Submittal Letter	X	X	X
	Approved FRD Issue Letter	X	X	X
	Approved FRD in Requirements Management Tool	X	X	X
4.2: SIC-2 Initial Architecture	Inspection Report	X	X	
	Design Review Report	X	X	
	Software Architecture Documentation	X	X	X
4.3: SIC-3 Risk Analysis	Risk Evaluation Documentation	X	X	X
	Software Hazard Analysis	X	X	
	Software Fault Tree analysis	X	X	
	Software Failure Modes and Effects Criticality Analysis	X	X	
	Findings Entered into Defect Tracking Tool	X	X	
4.4: SIC-4 Plan Incremental Builds	Integrated Build Schedule	X	X	X
4.5: SIC-5 Develop & Release Incremental Build	See below	See below	See below	See below
4.5.1: SIC-5.1 Develop	Approved Issued Updated FRD	X	X	X

SDPM Process Step	Process Deliverable	Flight Software	Ground Software	Test Bed
Detailed Requirements	Inspection Report	X	X	
	Design Review Report	X (if performed)	X (if performed)	
	FRD Submittal Letter	X	X	
4.5.2: SIC-5.2 Develop Detailed Architecture	Updated Software Architecture Document	X	X	X
	Inspection Report	X	X	
	Design Review Report	X (if performed)	X	
4.5.3: SIC-5.3 Detailed Module Design	Detailed Module Design Documentation	X	X	X
	Inspection Report	X	X	
4.5.4: SIC-5.4 Module Implementation	Module Source Code	X	X	X
	Inspection Report	X	X	
	Unit Test Cases and Results in CM	X	X	
4.5.5: SIC-5.5 Unit Testing	Reviewed Unit Test Cases	X	X	
	Unit Test Report	X	X	
4.5.6: SIC-5.6 Integration	Executables in CM	X	X	X
	Subsystem Test Cases and Results in CM	X	X	
	Defects Entered and Dispositioned	X	X	X
4.5.7: SIC-5.7 System Testing	Test Readiness Review Report	X	X	
	System Test Cases and Results in CM	X	X	
	System Test Report	X	X	
	Defects Disposed	X	X	

SDPM Process Step	Process Deliverable	Flight Software	Ground Software	Test Bed
4.5.8: SIC-5.8 Document & Release	Configuration Documentation	X	X	X
	Source Code and Executables	X	X	X
	Release Readiness Review Report	X	X	X
4.5.9: SIC-5.9 Independent V&V	Bettis Inspection Report	X	X	
	Bettis Test Report	X	X	
	Bettis Test Cases and Results in CM	X	X	
4.5.10: SIC-5.10a Defect Identification	Defects Documented in Defect Tracking Tool	X	X	X
4.5.11: SIC-5.10b Defect Disposition	Defect corrective actions implemented, verified, and closed out.	X	X	X
5.1: A1 Configuration Management [RESERVED]	Configuration Items in management tool checked in with revision history	X	X	X
	CCB Meeting Minutes Issued	X	X	X
5.2: A2 Requirements Traceability [RESERVED]	Traceability matrix between requirements, design, and code	X	X	
	Traceability matrix between requirements, test cases, and test results	X	X	
5.3: A3 Inspections [RESERVED]	Inspection Report			
	Findings			

SDPM Process Step	Process Deliverable	Flight Software	Ground Software	Test Bed
	entered into Defect Tracking Tool			
5.4: A4 Design Reviews [RESERVED]	Design Review Report			
	Design Review Closeout			
5.5: A5 Test Readiness Review [RESERVED]	Test Readiness Review Meeting Minutes			
5.6: A6 Release Readiness Review [RESERVED]	Release Readiness Review Meeting Minutes			
	Code, Executables, Documentation released for use			
5.7: A7 Software HA, SFTA, SFMECA [RESERVED]	Software Hazard Analysis			
	Software Fault Tree Analysis			
	Software FMECA			
	Findings entered into Defect Tracking Tool			
5.8: A8 Auditing/Self-Assessments [RESERVED]	Audit Report	X	X	
	Audit Response	X	X	
Other Processes/Deliverables [RESERVED]				

2.3 Software Life Cycle, Methodology, and Language

Software development for project Prometheus is performed using a phased development process with incremental builds. These incremental builds provide increasing levels of functionality until the complete functionality is provided in the final build. The incremental builds for JPL consist of software from all modules (Spacecraft, Mission, Reactor, and Ground). This creates the desire for the Reactor Module I&C software to provide incremental builds to fit with the JPL overall incremental builds.

2.3.1 Incremental Life Cycle

The incremental life cycle has been selected for NRPCT Prometheus Reactor I&C software development. This allows an incremental delivery of functionality as required by other Prometheus team members. The incremental life cycle is detailed in the SDPM.

2.3.2 Software Design Methodology

Once a software life cycle has been established, it becomes necessary to choose the approach to software design and implementation. Two major design paradigms were considered for this project: object-oriented design, and structured design.

Object-oriented analysis and design takes a view of the system that data and functions are intimately tied as a collection of objects, with each object having attributes and methods that may be invoked. Object-oriented design seeks to incorporate the principles of data abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. These principles were fostered in an attempt to overcome what was seen by some as limitations in more structured designs (global data, tight coupling between modules). The design process consists of identifying the objects for a system, establishing the attributes and methods for these objects, abstracting the objects to define the classes, and establishing the relationships between these classes. Once that has been completed, the software architecture can be developed, and then implemented, tested, and delivered.

Structured design is a more traditional approach to software development. Structured design begins by performing a functional top-down decomposition. This top-down development then apportions system functionality to various modules as laid out in the decomposition. Each module is then designed such that ideally there is only one start and end point. The functionality is then achieved through sequence, selection, and repetition structures. The system architecture is communicated through structure charts, data flow diagrams, flow charts, state diagrams, and other artifacts.

Both structured design and object-oriented design are capable of being used to capture and implement the functionality required of the Reactor Module. Structured design has been chosen as the method of choice since it has widespread experience and use in safety critical real-time embedded systems applications, has the most developed formal testing methods for verification, and has a very straight forward approach to design. Object-oriented design does have advantages with modularity and data-hiding, but has some disadvantages as well. Object-oriented designs need to avoid several concepts and coding constructs when being used in a real-time embedded application to ensure time response can be met and to avoid any software inspection burdens. Additionally, there is not a great deal of experience with object-oriented approaches when applied to safety critical real-time embedded software applications. These concerns made structured programming a more desirable approach for the extreme environment involved in the Prometheus Reactor Module.

2.3.3 Language Choice

A natural outgrowth of choosing a structured design methodology is the use of a structured programming language to complement the design methodology. The language chosen for use is ANSI C. This is a very mature and well understood programming language that is widely supported by compiler vendors. The C language has also been adopted by other organizations within the Prometheus team.

Other languages such as Java and C++ have a great deal of object-oriented features that must be avoided (and Java requires a virtual machine). Ada has been used by many embedded projects, but does not have as large of a developer base as C. FORTRAN has great power for applications heavy in numerical computation, but is not as well suited for embedded applications. The C language is a high level language that allows for very low level functionality when necessary.

3 WORK PRODUCTS

There are many work products defined by the PSMP. These products are enumerated here and related to specific Reactor Module software deliverables, or related deliverable items.

3.1 Software Development Plan

The software development plan represents the project specific embodiment of the principles and processes laid out in the PSMP. This document also lays out the schedule for software delivery for the various work products.

3.2 Requirements

Requirements are maintained in a requirements management database and issued as various versions of a Functional Requirements Document. Appropriate levels of control and traceability are applied at the database and document level.

3.3 Risk

As risks are identified they are documented and evaluated.

3.4 Risk Mitigation

Risk mitigation is identified in the same document that includes the risk.

3.5 Configuration Item

Configuration Items refer to any work item that is placed under configuration control. This includes requirements and requirements documents, source code, executables, test cases, and test results. The configuration management process identifies the system or systems used to control each configuration item.

3.6 Controlled/Quality Records

Controlled records or quality records refers to items that document specific things that do not change, or capture a moment in time, thus do not require configuration control because once created they do not change. These items include software inspection

reports, software testing reports, and the results from audits/self assessments. Once issued, these items are captured and retained through tools like ADSARS.

3.7 Software Item/Executable

The software executable is the final compiled and linked image that is uploaded to the supervisor or controller, it also refers to the Reactor Module Ground software. Executables may also consist of tools generated to support software development by the NRPCT.

3.8 Software Component

A software component consists of a collection of software modules that combine to achieve a specific goal. From a software architecture standpoint, an example might be the modules that combine to create the fault management system for an executable.

3.9 Software Unit

A software unit refers to a software module or compilation unit. This can be thought of as a source file with its associated header files. A module has been defined at the architectural level as providing a set of common data structures and functions to achieve a specific goal.

3.10 Software Class/Element

For object-oriented designs, a software class is simply the definition of a class, but for structured methods, the element refers more to a specific function or data structure.

3.11 Software Class/Element Instance

In object-oriented designs, the class instance refers to specific instantiation of objects. For structured methods, this is the creation of arrays, linked lists, or other data structures from the definitions provided by the software elements.

3.12 Software Item Delivery Record

The software item delivery record consists of the documentation provided with a software delivery that identifies versions and all source code and tools used to generate a specific version of the software.

3.13 Design View

The design view includes structure charts, flow charts, context diagrams, state diagrams, and other items necessary to convey the software architecture in a meaningful manner.

3.14 Design Document

The design document provides the overall software architecture, as well as details for each module to provide information necessary to understand the implementation of the software requirements.

3.15 Source Code

The source code is written to define each of the software modules laid out according to the software architecture.

3.16 Users Guide

The users guide provides the information necessary to be able to control the software. The guide would define all of the features supported by the Ground software to be able to communicate and send commands to the Reactor Module

3.17 Command Dictionary

The command dictionary provides the list of hardware or software commands that can be sent to the Reactor Module. For example, the command to start up the reactor would be listed in the command dictionary.

3.18 Telemetry Dictionary

The telemetry dictionary provides a listing of all of the telemetry that will be provided from the Reactor Module.

3.19 Flight Parameters

Flight parameters include various constants that might be set for the flight.

3.20 Flight Rules

Flight rules are any restrictions or guidance that the Reactor Module must obey. For example, not allowing a reactor startup while in the launch sequence of the mission would constitute a flight rule.

3.21 Review Materials

The review materials are the materials generated by the various design reviews that the software must undergo to achieve the necessary quality goals.

3.22 Training Record

The training record is a record of the training each person undergoes while learning to perform a specific role.

3.23 Test Procedure

A test procedure provides a step-by-step repeatable sequence necessary for a particular test.

3.24 Test Report

The test report is the documentation of the results of performing one or more test procedures.

3.25 Test Environment

The test environment is the hardware and software configuration, along with the configuration of any test tools used while performing testing. This is documented to be able to reproduce tests results.

3.26 Test Plan

The test plan is the overall plan used to qualify the Reactor Module software, and identifies the strategy and sequence of events to run the various test procedures.

3.27 Change Request

The change request is the means used to control changes to the software. Change requests go through a formal review process to determined when and how a change should be allowed.

3.28 Problem Report

A problem report documents the discovery of a defect (bug) in the software. Problem reports are reviewed in a manner similar to change requests to determine when and how to change the software to correct the defect.

3.29 Requirements Document

The requirements document provides all of the software requirements as a particular baseline. The requirements can be traced bi-directionally to the software implementation.

3.30 Interface Requirements Document (IRD)

The interface requirements document specifies the requirements on the interface between the Reactor Module and the Spacecraft Module.

3.31 Interface Control Document (ICD)

The ICD defines the data and commands that are transmitted over the interface.

3.32 Schedule Task

A schedule task is a specific task listed in the project schedule.

3.33 Schedule

A schedule is a listing of all of the various tasks, along with start and end dates for each task, and a relationship between the various tasks.

3.34 Budget

The budget identifies both manpower and funds necessary to develop the software.

3.35 Resources

Resources are the equipment, time, people, and money necessary to develop the software.

3.36 Work Breakdown Structure (WBS) Element

An item in the WBS, may be high level, or may be a low level item contained in a higher level item.

3.37 Work Breakdown Structure

The work breakdown structure is the relationship between all of the identified tasks necessary to develop the software. The WBS provides the relational framework for developing the budge and schedule.

4 TOOLS

There are many tools that are necessary to develop software for the SNPP. These run the gambit from requirements tools, to compilers, to test tools, to configuration management tools. Many factors affect the decision to use one set of tools over another. These factors include ease of use, overall availability, consistency with other software development efforts in the Prometheus project, and developer familiarity with the tools.

4.1 Requirements Management

Requirements management tools are a series of interfaces to a database that allow the developer to quickly document requirements, show how higher lever requirements are broken down, trace requirements to the code implementation and to the testing which validates them. Because this tool will play a large roll throughout the development process, its choice is particularly important. Within the Prometheus project there is also a desire to standardize on certain tools, because of the interrelated requirements with JPL and the spacecraft contractor using a standardized requirements tracking tool will be particularly important.

4.2 Configuration Control

Configuration control tools provide controlled access to a data repository. This controlled access allows for procedures such as approvals (CCB) and reviews to be completed prior to code submission. Another benefit is that multiple developers working on the same code cannot overwrite each others work. Conflict detection and merging is handled by the configuration control tool. The configuration control tool provides a historical record of all documents/code under its control. Ideally any document at any point in its history can be retrieved with little effort.

4.3 Problem Reporting

A problem reporting/defect tracking tool maintains a database of defects found either internally or externally. This tool allows testers to submit and track defects and allows cognizant developers to monitor these items. Ideally the tool will tie into the configuration management tool in order to tie fixes in the defect tracking tool to specific code submissions.

4.4 Compiler/Integrated Development Environment

The integrated development environment (IDE) provides a common interface to code development tools, debugging tools, compilers, and other tools. The IDE will typically include a code aware editor which provides color coding and automatic formatting functions. The common interface can significantly improve developer productivity.

4.5 Test Tools

A number of test tools will be used during different phases of development. These tools range from tools used by developers while writing code to full integration testing.

4.5.1 Debugger

The debugger is used examine the internal state of a running program. This allows a developer to quickly find and understand defects in the code. Debuggers are a significant factor in the productivity of developers and the quality of the code. Because of this, the functionality of the debugger is a driving factor in the IDE selection.

4.5.2 Code Coverage

Code coverage tools are used in testing to verify that all code paths are executed. There are several levels of code coverage ranging from simple statement coverage up to Modified Condition / Decision Coverage (MC/DC). For the Prometheus project MC/DC will be required. MC/DC requires that every point of entry and exit in the program be invoked at least once, every condition in each decision in the program be taken all possible outcomes at least once, and each condition has been shown to affect the decision outcome independently.

4.5.3 Unit Test

Unit testing tools provide a framework for developing efficient and repeatable unit tests. This often includes code generation support and automated test run facilities. By maintaining a database of previous test runs and results regression testing can be accomplished with a minimum of effort.

4.6 Other

The above list is not intended to be all inclusive. There is a wide variety of tools available that are designed to facilitate software development. Due to the desire among the Prometheus team members for commonality many of the tools will be selected through the Software Infrastructure Working Group. This should not rule out the selection of additional tools to meet the specific needs of the NRPCT.

5 REVIEWS

There are many NASA/JPL reviews related to the overall Prometheus project. There are also NRPCT internal reviews. These are identified here.

NASA/JPL reviews:
Project Mission System Review (PMSR)
Project Preliminary Design Review (PDR)
Project Critical Design Review (CDR)

There may also be a PDR and CDR on a Module basis.

NRPCT reviews:
Software Requirements Design Review
Software Architecture Design Review
Software Code Inspections
Software Test Case Reviews

NRPCT Software Development Plan

Appendix A – Traceability

6 APPENDIX A –TRACEABILITY [RESERVED]

6.1 PSMP Traceability

PSMP Requirement Number	*NSMP Section Number*
R-1.1.5-1 each SMP shall include a reference to the SDE's WBS	
R-1.1.5-1a Each SMP shall include a reference to a dictionary of the SDE's WBS elements.	
R-1.1.5-1b Each SDE's WBS shall be consistent with the software products' architectural 'implementation view'.	
R-1.1.5-2 Each SDE's WBS shall be consistent with the WBS for the SDE's parent organization.	
R-1.1.5-3 Each SDE shall maintain consistency of the WBS throughout the lifetime of the SDE.	

6.2 SS473 Traceability

6.3 NR Software Engineering Policy Traceability

6.4 KAPL Software Engineering Manual Traceability

CONCURRENCE/DESIGN CHECK FORM FOR DOCUMENT NO. SPP-67610-0007 Date: 7/30/05

DOCUMENT TITLE: **NRPCT Prometheus Reactor I&C Software Development Life Cycle, Design Methodology, and Programming Language Selection for Approval, and Draft Software Development Process and Plan for Information**

REFERENCES
a) JPL Document: Prometheus Project – Project Software Management Plan (preliminary), 982-00046, Rev. 0
b) Bettis Letter: B-REO(M)-CD-008, 3/17/05
c) KAPL Letter: ARP-68640-0196, 4/12/02
d) NAVSEA Letter: Ser. 08K/03-00484, 2/6/03
e) KAPL Letter: ARP-68640-0305, 9/2/04
f) KAPL Letter: FSO-64K20-04-143, 12/21/04
g) BPMI Letter: BPMI-ICS-PMP-00731, 3/11/05

ENCLOSURES
1. Space Electrical Systems Software Life Cycle, Methodology, and Language Choice for Prometheus Reactor I&C Software Development
2. NRPCT Reactor I&C Software Development Process Manual
3. NRPCT Reactor Module Software Development Plan

1. ADSARS: PERMANENT RECORD: Yes X No ___ Repository MFLIB Corporate Author: KAPL NR PROGRAM K

Key Words: Prometheus, Software, Life Cycle, Software Development Plan, I&C, NRPCT

Need to Know Categories	REP	SEP								
Available Sites:	PRNR	PRNR								
Design File Location(s)										

2. DESIGN CHECK

Type of Check	Signature(s)	Comments: (Including Reference to Check Document If Appropriate)
A. No check considered necessary		
B. Check vs. previous results/issues		
C. Checked calculations made		
D. Checked computer input and/or output		
E. Computer Programs approved/qualified		
F. Performed independent audit		
G. Spot checked significant points		
H. Reviewed methods used		
I. Reviewed results for reasonableness		
J. Comparison with test data		
K. Reviewed vs. drawings		
L. Verified procedures		
M. Technical content reviewed	[signature]	
N. Management verification of adequate review by others		
O. Performed Lessons Learned Search		
P. Used Measurement Uncertainty Methods		
Q. Other Checks (Describe)		

3. CONCURRENCE REQUIREMENTS: Indicate signatures required by X:

___	SPP MANAGER	___	ADVANCED CONCEPTS	___	FLUID DYNAM
___	NUCLEAR ENGINEERING	___	SHIELDING	___	STRUC. ENGR
___	REACTOR TH/MECH DESIGN	___	REACTOR SAFETY	___	DRAFTING
___	REACTOR EQUIPMENT	___	TO	X TAH for M. Katzer per telecon	QA
___	SPP MECHANICAL	___	RSO	___	OTHER
___	SPP ELECTRICAL	___	FSO	X TAH for D. Robare per telecon	BETTIS
___	FINANCE	___	MDO	___	BPMI
___	PROJECT OFFICE	___	ARP	X DRS	ADMIN REVIEW
___	ENERGY CONVERSION				

Cognizant Manager [signature]
(Must Be Subsection or Higher for External Letters)

4. AUTHORIZED CLASSIFIER: **Reviewed By:** [signature] **CLASSIFICATION:** UNCLASSIFIED

5. RELATED SUBJECTS:

UTRS Implication (Y/N)	N	Commitment Made (Y/N)	N	Commitment Complete (Y/N)	N
Safety Council Review (Y/N)	N	Design Basis Info. (Y/N)	N	UTRS Doc. #	N/A
		Design Review (Y/N)	N		

PRE-DECISIONAL – For Planning and Discussion Purposes Only

6. Distribution:

NR
TH Beckett, 08B
S. Bell, 08I
DI Curtis, 08S
DE Dei, 08A
AJ Demella, 08H
JE Eimes, 08F
RA Glas, 08H
MW Henneberger, 08K
SR Kauffman, 08Z
JM Kling, 08Y
JM Mckenzie, 08U
TJ Mueller, 08R
JP Mosquera, 08C
JW Moy, 08M
MD Natale, 08I
WJ Pollock, 08T
TN Rodeheaver, 08I
SJ Rodgers, 08E
CH Oosterman, 08C
SJ Trautman, 08V
RA Woodberry, 08G

KAPL
JM Ashcroft, 132
CF Dempsey, 111
KC Loomis, 132
DF McCoy, 111
H. Schwartzman, 132
SA Simonson, 081
MJ Wollman, 111
M Ryan, 132
T Hamilton, 132
D Schroeder, 132
B Robinson, 132

LB

SNR

DJ Potts
GM Millis
D. Clapper
W. Leahy
S. Cramer

Bettis
CW Clark, 01C/SE
CD Eshelman, 36E/SE
DP Hagerty, 38D/SE
RC Jewart, 01C/SE
DR Riley, 05P/MT
CA White, COB1/QMA
MJ Zika, 01C/SE
DJ Robare, 43T/SE
TS Blazeck, 43T/SE

BPMI-P
SD Gazarik
RF Hanson

PNR
J. Andes
RJ Argenta
JF Koury
G. White

BPMI-S
F. Barilla