University of Alaska Fairbanks
Engineering and Science Management

ESM - 694
Project Report

An examination of the architecture, cost effectiveness, and management decisions leading to the development of a new Satellite Ground Station Control software system

by
Norman Cushing
Presented Dec 13th, 2001

Table of Contents

1 Overview

This project will explore the management decision to design and develop a new software system for controlling and managing satellite ground station equipment.

Areas to be examined include the old system, the new system, areas that typically change once a system has been fielded, the cost to modify the old system, the cost to develop the new system, the cost to modify the new system, and estimated payback projections for the new system.

2 Goal of New Architecture

The goal of the new architecture is to obtain a cheaper system as measured by a reduction in sustaining and enhancement costs. Sustaining cost being defined as the cost associated with fixing bugs and maintaining the life of the product by being able to migrate it to new computer systems and operating systems as older systems and operating systems are obsolesced by the vendor. Enhancement cost being defined as the cost associated with modifying the software to perform tasks and control equipment that were not included in the initial requirements nor in the initial release and fielding of the software.

3 Overview of the Old Architecture

This section describes the architecture of the old system. It will discuss the types of software development constructs and methodologies that have been used. It will examine the effectiveness of these techniques.

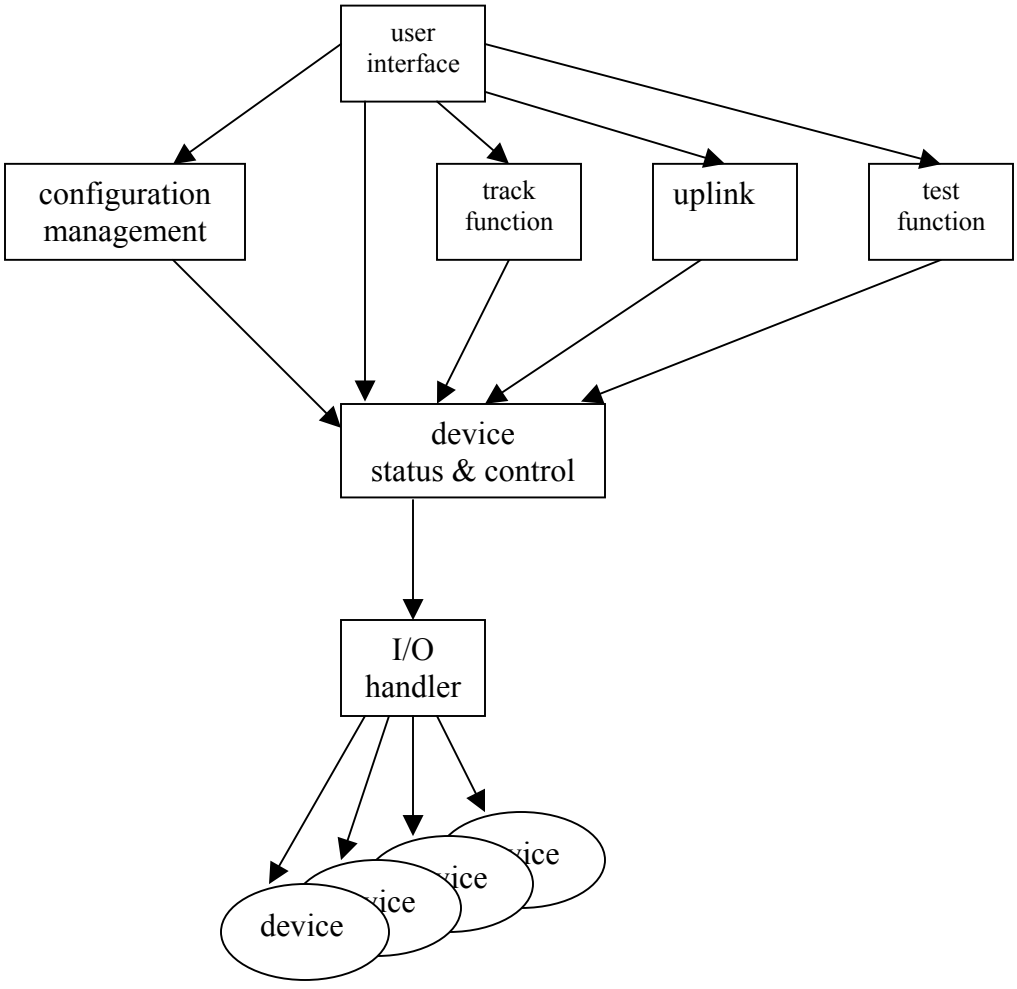
3.1 Background

The old system is approximately 15 years old. It has been delivered to approximately 25 customers. To varying extents, the software and hardware have changed for each

customer. Following the initial release, each customer has received approximately 3 additional software releases with some customers having received 24 releases.

A typical system consists of 48 software programs (executable applications) that work as a team to carry out the functionality of the system. These executables are constructed from 952 source code files. Of these 952 files, approximately 430 are shared files. A shared file being a header file (381 files) in which data types, variables, and function calls are defined, a function file (49 files) that are part of a library consisting of commonly used utilities.

This diagram shows the major components of the old architecture. The arrows represent the flow of control from the user interface to the device. Arrows to show the flow of status from the device would point in the opposite direction.



3.2 Variations in Files

Over the years the old architecture has under gone many small to significant changes for different customers. Each of these changes has caused a change in the source code. These different versions of the source code are managed by the company's configuration management system. Today it is difficult to discuss what exact function a particular software module performs. A meaningful discussion can only begin once the customer and release of the software are identified. Generally, because of the many changes that have occurred over the years, the individual source code files must be examined to get an exact understanding of what any one permutation of the software will perform. The large number of permutations of the various files and functional logic they contain, adds to the complexity of maintaining the old architecture.

3.3 Old Architecture Technologies

The old architecture was implemented using the C language. It was originally developed for the UNIX platform and has been ported to NT. It utilizes sockets, pipes, and message queues, and shared memory for inter-process communication. The user interface has been implemented using the X-Motif library. The UIM-X software tool was original used to develop the user interface but has since been abandoned. Reports and graphs have been implemented using the XRT drawing library and hard-coded reports.

3.4 Areas that are Expensive to Change and Why

This section will describe the type of changes that have occurred to the old system following its conception, design, and fielding.

3.4.1 Types of Changes and Areas Affected

During the life of the old software system it has been modified to:

- Control additional hardware devices added to the system to enhance capability.
- Control new devices that are replacing obsolete devices.
- Handle different formats of input data.
- Generate new types of reports.

Many of these changes occur to support new customers that need systems that are just like a previous system but with a few unique changes. This latter type of change is quite common and accounts for most of the software maintenance and enhancement activities. As an example, a typical device change may stem from a change in antenna pedestal such as changing from an x-y to an azimuth/elevation or even a 3-axis configuration that utilizes a train axis, (a train axis allows the pedestal to be tilted to avoid a hole at the zenith during high elevation passes). Pedestal changes typify the type of change that is important to the customer. The x-y pedestal is cheaper but has less performance than a 3-axis configuration that because of its greater complexity cost more.

A typical change to add a new device generally involves changing many of the programs that constitute the system. If a new device is added it is possible the following programs will require a change:

- Device Status and Control,
- Configuration management,
- I/O handler,
- User interface,
- Any number or all of the higher-level activities that utilize the device to perform a higher-level activity, (i.e., such as tracking a satellite, transmitting commands to a satellite; or any manner of system characterization tests, such as bit error rate, gain over temperature, etc.).

3.4.2 Why Changes are Costly

A customer order for a new system that is very similar to an existing fielded system but differs in some of the devices has historically required an effort of 300 to 400 man-hours

of software development changes before it is ready to be fielded. A typical change to add new or to modify an existing device has required changing and/or creating 20 software files.

Some systems are quite unique and only exist as a one-of-a-kind configuration of software and hardware. These systems can be modified at the plant but can only be tested in the field. During the initial development of the system, software simulators of the hardware devices were not constructed. If software simulators of the hardware devices had been constructed then the software could be tested in the plant. This would eliminate the need to send a software engineer into the field to finish the software development effort.

4 Cost of changes to old architecture

This section will explore in detail the cost associated with making changes to the old architecture.

4.1 Modifying Configuration Management to use ASCII Configuration Files

Configuration files are used to set up and initialize the hardware and software prior to a satellite tracking activity. A configuration file will contain the frequency the satellite will use to broadcast its signal, the bandwidth of the signal, the type of search pattern the antenna should perform to locate the signal, the transmit data rate, etc. In the old architecture the default values for configuration file items has been hard-coded in the source code. For a typical system there are 37 source code files with hard-coded values. These source code files contain from 1 to as many as 63 default configuration items, with the average file containing 13 items. At system startup this default information is written to a disk file that is named configuration zero. As many as 999 additional configuration files can be created, by using the user interface to change the default values to specific values that are required for a particular satellite mission or system test, and saved to disk and given a unique configuration file name. Before a satellite mission begins the

operator creates a satellite specific configuration file. During a satellite pass, when the station will be used to track a satellite, the operator selects the previously created and stored file and applies it to the hardware.

Customers have requested that configuration files be human readable and editable. Human readable and editable files would allow files to be shared between fielded systems and would allow for the ground support staff at each location to verify the content of a configuration file without having to apply the configuration to the system's hardware.

Configuration data files were implemented as binary blobs. A binary blob is a group of data items that are managed as a single item and are left in the native binary number representation of the computer. In the old architecture each data item in the binary blob is represented in 4 bytes (32 bits) of data, regardless of data type (integer, real, boolean, enumeration). Character strings are not allowed because they typically cannot be represented in as few as 4 bytes.

Viewing a configuration file with a dump program is not very meaningful to the average customer operator. To interrupt the data requires translating the binary blob into human readable form. This can be done with lots of proprietary knowledge (access to the source code) and a conversion algorithm.

The algorithm used to store this data was developed to minimize disk space and data handling time. The algorithm was implemented in the mid 1980s when disk space was much more expensive and less abundant, and when computer processing speeds were slower. Therefore, consideration for minimizing the usage of these resources was prudent.

Modification of the old architecture to implement ASCII configuration files was undertaken during the spring of 1999. An estimate of 3 man months of effort was calculated and funding was obtained. At the end of the 3 man months when the budget

was exhausted the development effort stopped and has never been finished. The current estimate to complete the work is set at an additional 3 man months of effort.

Why was it so costly to convert to ASCII configuration files and why will it be so costly to complete the effort? To answer this question we need to examine the old architecture and the configuration management scheme.

In the old architecture a typical hardware device configuration data structure looks like this (constructed of integer, float, boolean, and enumeration data types):

```
typedef struct {
      INT                     fine;
      INT                     coarse;
      phase                   clock_phase;
      channel_mode            data_channel1;
      channel_mode            data_channel2;
      encode_switch           encoder_switch;
      imped_switch            impedance_switch;
      mod_switch              modulator_switch;
      INT                     qpsk_format;
      inp_mode                input_mode;
      mod_source              modulation_source;
      on_off                  config;
} sa924_6;
```

The assignment of the default values to the data structure looks like this:

```
static sa924_6 default_sa924_6  =
{
      0,                /* fine */
      0,                /* course */
      falling,          /* clock_phase - 180 */
      inverted,         /* data_channel1 - inverted */
      noninverted,      /* data_channel2 - non-inverted (normal) */
      pass,             /* encoder_switch - pass (enabled) */
      ohms50,           /* impedance_switch - 50 ohm */
      qpsk,             /* modulator_switch - qpsk */
      mode_1,           /* qpsk_format - mode 1 */
      serial,           /* input_mode - serial */
      int_src,          /* modulation source */
      on                /* config */
};
```

Each data item that is represented by an enumeration must have the enumeration defined:

```
typedef enum {
      rising  = 0,
      falling = 1
} phase;
```

```
typedef enum {
      inverted = 0,
      noninverted = 1
} channel_mode;

typedef enum {
      pass = 0,
      bypass = 1
} encode_switch;

typedef enum {
      ohms50 = 0,
      ohms75 = 1
} imped_switch;

typedef enum {
      uqpsk4_1 = 0,
      uqpsk5_1 = 1,
      qpsk = 2,
      bpsk = 3
} mod_switch;

typedef enum {
      mode_1 = 1,
      mode_2 = 2
} s_band_qpsk_format;

typedef enum {
      serial = 0,
      parallel = 1
} inp_mode;

typedef enum {
      ext_src = 1,
      int_src = 2
} mod_source;

typedef enum {
      off = 0,
      on = 1
} on_off;
```

All of this information must be converted into new software that uses character strings. Additionally, the field name, data type, range and units of all data items must be represented to the user so that correct interpretations of the files contents can be made without the need to apply the file to the hardware.

Another added expense has been incurred because of the way the software was managed and changed over the years. During previous changes to the software, minimal effort was

made to have a single code file work with different types of hardware. Therefore, multiple files exist that perform similar functions but for different types of hardware. This has forced the underlying configuration data structures to be unique across the 25 plus fielded systems. Having the same underlying data structure across all deployed systems is an important requirement for NASA that has 5 stations using this code as a baseline. Of those 5 systems only 2 use the same exact source code. The reason for this is because only 2 of the sites have identical hardware. Although most of the software is the same for the 5 sites, the software that configures and controls devices is unique when different devices are used for the same or similar purpose.

4.2 Adding a New Device or High Level Function

Adding a new device to the old architecture involves several of the major components.

- Configuration Management
- Device Status and Control
- User Interface
- Message Queues

Because the old architecture depends upon data structures that are common to all software modules (device configuration data, and message queue structure) a change to add or modify an existing device can perturb the entire suite of software programs. The position of data in these data structures is dependent upon its neighbors. For any specific data item, if a neighbor is removed or added the position of all downstream data items shifts. Therefore, making a change to the structure forces all software modules to be rebuilt.

The configuration management program contains information about each device in the system. It reads and writes configuration information that is used to setup the devices and to maintain system characterization test parameters. As mention early, configuration data has been implemented as binary blobs. Therefore, if a new device is added to the

system or the configuration data structure for an existing device is changed, all existing configuration files must be modified. The change involves moving existing data items to make room for the new item. The file is structure so that like devices are together. Therefore, new data items and devices are not appended to the end of the file.

The device status and control program is also dependent upon a set list of devices. If a new device is added the identifiers, (implemented as an enumeration), for all devices may get modified. A change of this type requires all programs to be rebuilt so that all are using the same identifiers. Additionally, the status and control program uses a jump table to dispatch the device control and status commands to specific functions that contain the logic to communicate and understand the message traffic to and from the device. These functions are also managed in a data structure that is positional dependent. Therefore, the addition or removal of a device and its associated functions causes the entire device status and control logic to be rebuilt.

The user interface is a massive all encompassing program that uses X-motif. Because this program directly has knowledge of each data item in the system that must be collect from or displayed to the user it knows about everything in the system. If a new device is added or removed, the user interface is always impacted significantly. Some of the reason for this impact is inherited in the nature of how data is displayed on the user interface. The user interface is provides functions to configure hardware and control hardware and it displays hardware status at different levels of aggregation. Therefore, a change in a device can impact several unrelated sections of the program. Similarly to the other system components the user interface uses enumerations and data structures that contain information about each device and function in the system. A change to these data items generally causes a change across the entire program.

For inter-process communication the system uses message queues. These message queues have been implemented using low-level, operating system dependent, constructs. The message transmitted across the sockets contains an identifier of the originating and destination programs, a message type, and then a message dependent data structure.

Adding or removing a program to the suite of system software will cause a change in the message queue data structure. This will also cause a rebuild of the entire software suite.

As devices and high level tests and functions are added, removed, or modified many global data structures are affected. Modification of these structures causes the entire software suite to be modified. Because changes cannot be isolated to a small isolated block of software it is common to make a change in one area that causes an unexpected and unwanted change in another section. Generally, the intended change is not very significant, but managing the implementation so that its side effects are controlled is what takes the most resources and discipline from the software team.

5 Overview of New Architecture

This section describes the architecture of the new system. It will discuss the types of software development constructs and methodologies that have been used. It will examine the effectiveness of these techniques.

A major goal of the new architecture was to correct some of the limitations with the old architecture that have reduced its flexibility and cost effectiveness when changes are required after it has been fielded. For instance, for new customer orders, the new system must be re-configurable and able to support different satellite missions with minimum changes and effort. Additionally, the new system must be capable of full factory testing there-by eliminating the need to have a software engineer complete the software testing in the field.

5.1 Architectural Components

The new architecture has been described as a bowl of jelly beans. A jelly bean is an executable program that works as a team member with other jelly beans to carry out the functions that are performed by the software system. From this bowl of jelly beans there are the following types of components:

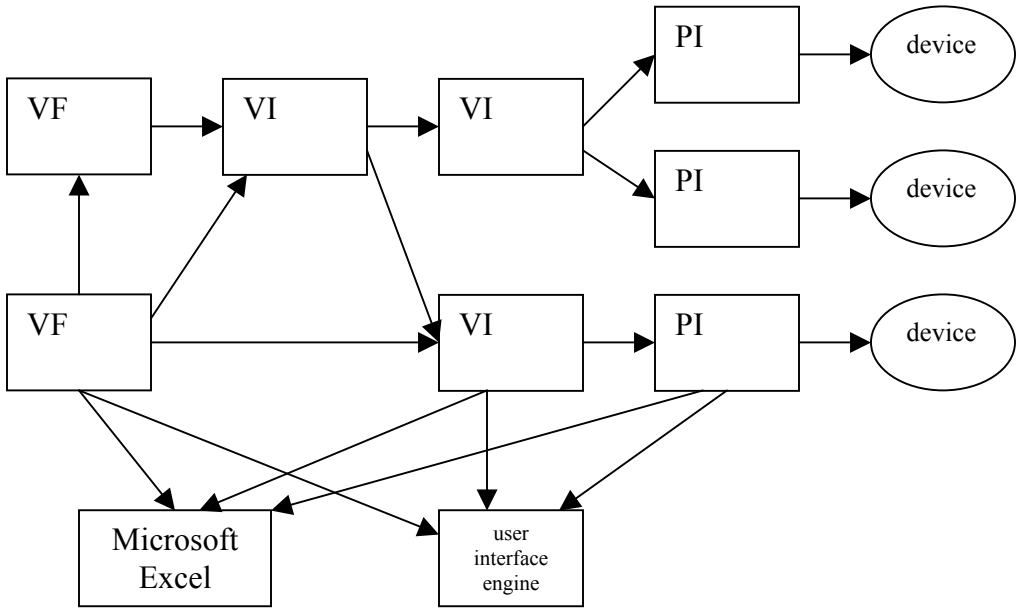
- Physical Instruments
- Virtual Instruments
- Virtual Functions
- Infrastructure

Physical instruments interact directly with the hardware. All physical instruments of a particular class or category of device, based on device usage and capability, export a common external interface. This common interface allows higher-level software to communicate with any physical instrument in the class without the need for software changes. Therefore, if a device becomes obsolete and is removed from the system, and a new device replaces the old device, all that is needed to use the new device is a new physical instrument layer of software that exports the common external interface for a device of the class that it is replacing.

Virtual instruments encapsulate, and status and control physical instruments of a particular class. A virtual instrument is an abstraction of the class of physical instruments that it represents. The purpose of the virtual instrument is to present a common interface for the class of devices to higher-level software. Presenting an abstracted class interface allows the physical instrument and hardware device to be changed without any need to change any software above the virtual instrument.

Virtual functions implement high level activities that will be performed by the system such as satellite tracking, satellite commanding, and system characterization tests such as bit error rate, gain over temperature, etc.

Infrastructure functions implement higher-level activities such as data collection, error reporting, report generation, etc.



This drawing shows the relationship between the various types of Jelly Beans that can make up a typical system. These relationships can be expanded to include more complex relationships as needed to construct a desired system. The arrows represent the direction that communication links are initiated.

Jelly Beans communicate using two CORBA [Henning 1999] mechanisms. The use of CORBA allows the system components to be distributed among different computers including machines on different but connected networks. Thus the machines can be collocated with the equipment or remotely located in a more central and convenient location. Also, devices can be built that support network and CORBA interfaces, thus eliminating the need to have a dedicated computer to control the devices. The first

mechanism is a direct remote procedure call (rpc). An rpc allows one program to call a function in another program. The Jelly Bean that offers the rpc function provides an interface description of the rpc that is utilized by those objects that are interested in utilizing its function. Using this mechanism a physical instrument can provide a virtual instrument an explicit method for telling the physical instrument to control its device or to perform a specific function. Each physical instrument of a specific class type will provide the same rpc interface. This will allow different physical instruments of the same class to be interchanged without affecting higher-level software (virtual instruments). This same mechanism is utilized by all higher-level software components (virtual instruments and functions, and infrastructure) to explicitly communicate among each other.

The second communication mechanism is an event channel. An event channel allows any Jelly Bean to subscribe to information (telemetry) from any other Jelly Bean. A subscribing Jelly Bean must know what telemetry is available and which Jelly Bean will publish the telemetry. The publishing Jelly Bean does not know or care who is subscribing. The publishing Jelly Bean provides an interface description of its telemetry that is utilized by those objects that are interested in subscribing. The description defines the content, and data structure of the telemetry, there-by providing the necessary information to the subscriber so that it can ingest and utilize the telemetry. A subscriber cannot demand telemetry from the publisher. Telemetry is only available to the subscriber when the publisher publishes. All subscribers will receive telemetry at the same time.

The user interface has been implemented in 2 pieces consisting of the engine and the description. The engine has been implemented in JAVA [Lemay 1999] and is the workhorse of the interface. The JAVA implementation allows the user interface to be displayed to any network location that has a browser. This allows the status and control of the system to occur both locally and remotely. It is an application program that receives the description of the windows and buttons and their content, and paints them on the CRT. It also accepts user input and hands it to the Jelly Bean that will process the

data. The description of the interface resides with each Jelly Bean that requires a user interface. This allows an overall smaller user interface engine. And most important, it provides a user interface that is not dependent upon the application that it presents to the world. As Jelly Beans come and go, because they are removed or added to the system, the user interface changes accordingly. In other words, the user interface content and look and feel is controlled by the objects that need a user interface and not by one large and encompassing program as was done in the old architecture and is common for most applications. Because the user interface is a part of each Jelly Bean, this mechanism eliminates the need for software development to change the user interface as Jelly Beans are changed.

To generate a report or graph each Jelly Bean formats its output data for ingestion by Microsoft Excel. Once the data has been filtered and a graph generated by Excel it is exported to an HTML file where it can be displayed using a browser. These technologies allow the data to be viewed from any network location.

5.2 Software Technologies of the New Architecture

This section will discuss the major software tools that will be utilized during the development process. These technologies are being employed in an attempt to gain more efficiency from the software development team, to increase the quality of the final product, and to manage change.

5.2.1 Rhapsody

Rhapsody [Rhapsody 2001] is a C++ object-oriented fully integrated Visual Programming Environment in which the developer can analyze, design, implement and verify the behavior of complex software applications. This tool implements the Unified Modeling Language [Douglas 1999].

The goal for purchasing Rhapsody is to reduce the software lifecycle costs for the new architecture. Rhapsody will aide in meeting this objective because of its development paradigm. The Rhapsody paradigm focuses the development team on constructing and

maintaining a design model, with the Rhapsody tool generating source code from the model.
The cost of Rhapsody for the 14-member software development team for the 1st year is $228,000, which includes the cost of the tool, training, and a 1-year usage license for each engineer.

Chart 1 shows a simple cost recovery analysis using a 0% internal rate of return (IROR).

Chart 2 shows that an IROR of 10% or greater can be achieved between years 1 and 2, if the use of Rhapsody can produce a labor efficiency of 10%. An increase in labor efficiency allows for a reduction in staff that directly contributes to reducing the labor expense. The calculations assume the Rhapsody tool will not have any salvage value at the end of its useful life.

Chart 3 shows that to realize an IROR of 10% by the end of year 3, the Rhapsody tool must provide for an increase in labor efficiency of 5.38%. The increase in labor efficiency will allow for a reduction in staff that contributes to a reduction in labor expenses.

For charts 2 and 3 these higher levels of efficiency are necessary because the initial costs of Rhapsody are high while the benefits occur in future years.

Chart 1

Simple Cost Recovery Analysis of Rhapsody Software Development Tool

(All costs are in thousands of dollars)

					Cumulative Cost of Efficient Labor and Tool Expense								
Year	Rhapsody Expense	Cumulative Rhapsody Expense	Labor Expense	**Cumulative Labor Expense**	Total cost with 0.0% Efficiency	Total cost with 1.0% Efficiency	Total cost with 2.0% Efficiency	Total cost with **3.0%** Efficiency	Total cost with **4.0%** Efficiency	Total cost with **5.0%** Efficiency	Total cost with **6.0%** Efficiency	Total cost with 10.0% Efficiency	Total cost with **11.0%** Efficiency
0	228	228											
1	30	258	2263	**2263**	2491	2469	2446	2423	2401	2378	2355	2265	**2242**
2	30	288	2263	**4526**	4814	4769	4724	4678	4633	4588	**4543**	4362	4316
3	30	318	2263	**6789**	7107	7039	6972	6904	6836	**6768**	6700	6428	6361
4	30	348	2263	**9053**	9400	9310	9219	9129	**9038**	8948	8857	8495	8405
5	30	378	2263	**11316**	11694	11580	11467	11354	**11241**	11128	11015	10562	10449
6		378	2263	**13579**	13957	13821	13685	**13549**	13414	13278	13142	12599	12463

Question: This table answers the question: At what efficiency level is the cost of the Rhapsody tool recovered?

The right side of the matrix shows the year and labor efficiency level that must be obtained from the Rhapsody tool in order to recover the cost of the tool.

The tool pays for itself when the combined cost of the tool expense and increase in labor efficiency are less than or equal to the labor expenses without the tool.

Answer: The BOLD items on the right show the efficiency level, for a given year, that must be achieved for the cost of the tool to be recovered. This occurs when the "Running Cost of Efficient Labor and Tool Expense" is less than "Running Labor Expense".

Parameters: 14 Seats for 14 Engineers, with Kickstart Training and Consulting

Annual labor cost per SW engineer (in thousands) = 162

Number of SE engineers = 14

Chart 2

This chart shows the IROR for estimated useful lives of 1 to 6 years if a 10% level of labor efficiency is achieved with the use of Rhapsody

Year	0	1	2	3	4	5	6
Rhapsody purchase	$ (198,000)						
Rhapsody licensing	$ (27,859)	$ (27,859)	$ (27,859)	$ (27,859)	$ (27,859)	$ (27,859)	
Labor savings using Rhapsody		$ 226,300	$ 226,300	$ 226,300	$ 226,300	$ 226,300	$ 226,300
Annual cash flow	**$ (225,859)**	**$ 198,441**	**$ 198,441**	**$ 198,441**	**$ 198,441**	**$ 198,441**	**$ 226,300**
IROR	**N/A**	**-12%**	**47%**	**70%**	**79%**	**84%**	**86%**

$	2,143	Rhapsody annual licensing cost per engineer
$	161,643	Annual labor expense per engineer
	14	Number of engineers assigned to project without Rhapsody
	12.6	Number of engineers assigned to project if Rhapsody provides 10% increase in labor efficiency
$	2,263,002	Labor without Rhapsody
$	2,036,702	Labor using Rhapsody
	13	Number of Rhapsody licenses
		(must be a whole number greater or equal to the number of engineers assigned to the project)
	10.00%	Labor efficiency with the use of Rhapsody (in this example)

This chart shows that an IROR of 10% or greater can be achieved between years 1 and 2, if the use of Rhapsody can produce a labor efficiency of 10%. An increase in labor efficiency allows for a reduction in staff that directly contributes to reducing the labor expense. The Rhapsody tool will not have any salvage value at the end of its useful life.

Chart 3

This chart shows the number of engineers and the labor efficiency that must be realized with the use of Rhapsody, if the tool is to provide an IROR of 10% by the end of year 3.

Year	0	1	2	3	4	5	6
Rhapsody purchase	$ (198,000)						
Rhapsody licensing	$ (30,002)	$ (30,002)	$ (30,002)	$ (30,002)	$ (30,002)	$ (30,002)	
Labor savings using Rhapsody		$ 121,698	$ 121,698	$ 121,698	$ 121,698	$ 121,698	$ 121,698
Annual cash flow	**$ (228,002)**	**$ 91,696**	**$ 91,696**	**$ 91,696**	**$ 91,696**	**$ 91,696**	**$ 121,698**
IROR	**N/A**	**-348.6%**	**-13%**	**10%**	**22%**	**29%**	**34%**

$ 2,143	Rhapsody annual licensing cost per engineer
$ 161,643	Annual labor expense per engineer
14	Number of engineers assigned to project without Rhapsody
13.2	Number of engineers assigned to project if Rhapsody provides labor efficiency
$ 2,263,002	Labor without Rhapsody
$ 2,141,304	Labor using Rhapsody
14	Number of Rhapsody licenses (must always be a whole number that is equal to or larger than the number of engineers assigned to the project)
5.38%	Desired labor efficiency with the use of Rhapsody

The Rhapsody tool will not have any salvage value at the end of its useful life, which has been chosen to be 3 years. This chart shows that to realize an IROR of 10% by the end of year 3, the Rhapsody tool must provide for an increase in labor efficiency of 5.38%. The increase in labor efficiency will allow for a reduction in staff that contributes to a reduction in labor expenses.

5.2.2 Make Files (Cross platform)

Operating system independent make files will be utilized to eliminate any problems with porting the software between NT and UNIX. This is an important issue so that a single version of the software is all that is required to support customer that have either an NT or UNIX requirement.

5.3 Areas expected to change (modified, expanded, and enhanced) over time.

This section will describe the type of changes that are expected to occur to the new system following its conception, design, and fielding.

5.3.1 Types of Changes and Areas Affected

The expected types of changes to the new system are similar to the changes performed on the old system. New customers will want a system that is similar but different to an already fielded system. Some customers will want to make changes to their system to allow for expanding their mission capability.

These changes will cause new devices to be added to the system and old devices to be removed. Additional tests and high level functions will be added. Additionally, more complex management tasks will be added that will allow the system to become more autonomous and provide more redundancy.

5.3.2 Why this is Architecture is Cheaper to Change

The new architecture should be cheaper to change because it utilizes the concept of information hiding and functional partitioning. In other words, a Jelly Bean does not affect its neighbors. A Jelly Bean is self-contained and can run independently of other

components that are not directly necessary for it to perform its task. Changes to the Jelly Bean are isolated to the Jelly Bean and no other system components.

6 Estimated Cost of Changes to New Architecture

This section will explore in detail the cost associated with making similar changes to the new architecture.

6.1 Modifying the Content of Configuration Files

Today, larger disks and faster CPU resources are available and cheap. Therefore, a different algorithm can be developed that handles data that is human readable and editable. The new algorithm must know how to translate between binary and ASCII formats, and it must know how to label the data so a human can distinguish and interrupt each data item. There are at least two designs that can be implemented.

A solution is to implement a new feature that allows the system to report (print or save to disk in ASCII format) a configuration file. This solution would continue to use binary blobs but its internal data structures would be modified to contain extra fields that the system can use when reporting configuration file content. The user interface would still be the only way to edit a configuration file. The extra fields might include items such as:

- data field name
- data value
- data units
- used with this system (This is a boolean value that determines if this data item will be used or ignored by the system being configured by this file. This is useful for customers such as NASA which has many similar (but different) systems scattered around the world perform the same satellite mission. This would allow those systems to share configuration files.)

Another solution is to change the configuration files from binary blobs to ASCII formatted files that utilize the XML format. This type of format can be used to contain

any number of parameters that describe the data item. An XML configuration file example is:

```
<XML>
        <UPCONVERTER_TYPE_1_UNIT_1>
                <USED_IN_THIS_SYSTEM>TRUE</USED_IN_THIS_SYSTEM>
                <UPLINK_FREQUENCY>
                        <DATA_TYPE>FLOAT</DATA_TYPE>
                        <SIGNIFICANT_DIGITS>8 </SIGNIFICANT_DIGITS>
                        <VALUE RANGE=1000.0000,2500.0000>1245.6872</VALUE>
                        <UNITS>KM/S</UNITS>
                        …
                </UPLINK_FREQUENCY>
                <UPLINK_BANDWIDTH>
                        …
                </UPLINK_BANDWIDTH>
                …
        </UPCONVERTER_TYPE_1_UNIT_1>
        …
<XML>
```

This type of file is easy to parse, and is human readable and editable. Although human readable, it is slightly hard to read until one gains familiarity with the format and special content characters. For the computer, it is slower to read and write, because it is constructed using many more bytes of data, and each data item must be examined to determine what it is, and how it should be interrupted. With today's faster computers the moderate amount of additional bytes is not much of an issue.

6.2 Adding a New Device or High Level Function

For the new architecture, as in the old, it is important to define what is meant by "to add a new device". Recall that the new architecture can be considered a bowl of heterogeneous Jelly Beans with each Jelly Bean representing a different capability. To deploy a new

system the correct Jelly Beans are selected from the bowl and assembled into the system. The Jelly Beans selected to be part of the new system are selected based on their capability to fulfill the requirements of the system.

With this understanding there are two possibilities for adding a new device to the system. Either a Jelly Bean which implements the functionality of the new device already exists in the bowl of Jelly Beans and therefore can be selected, or if the type of device to be added is entirely new, then the Jelly Bean does not exist and must be created.

The process of creating a new Jelly Bean follows a typical software development effort. However, with the new architecture the development effort is isolated to the new module and modifications to two or three higher level and existing XML configuration files. Changes to the higher level files provide communication mechanism information to the other system components that will communicate with the new Jelly Bean.

The Stovokor [Manison 2001] XML file is the system's main configuration file. It contains an entry for every Jelly Bean in the deployed system. This file informs the other system components that the new device has been added to the system and how to communicate with it. Here is an example of the information that must be added to the Stovokor XML file when a dehydrator physical instrument device is added to the deployed system.

```
<MODULE LOCAL="Yes" USE_FACTORY="Yes" NAME="DEHY_PI_1"  EXECUTABLE="factory.exe">
   <TYPE>DehydratorC_ETI</TYPE>
   <FACTORY_NAME>DEHY_PI_FACTORY</FACTORY_NAME>
</MODULE>
```

The NAME is the field other Jelly Beans will use to communicate with the new Jelly Bean. Also, the NAME field is the name of the Jelly Bean specific system setup file (see below).

Software development of the new device's module entails creating a new Physical Instrument executable program (the functional part of the new Jelly Bean) that will perform the function of controlling and monitoring the device.

Additionally, three XML formatted data files must be written as part of the new Jelly Bean. These are the device specific system setup file, defaults file (device configuration file), and screens file (user interface look and feel). The system setup file describes the device's user interface name, the name of the screens file, the name of the defaults file, and the type of communication that is required between the software and the physical device.

Here is an example of the device specific setup file for the dehydrator.

```
<XML>
   <GUINAME>Dehydrator Controller PI</GUINAME>
   <SCREENS>DEHY_ETI_PI_Screens</SCREENS>
   <DEFAULTS>DEHY_ETI_PI_Defaults</DEFAULTS>
   <COMMADAPTER>CASOCKET</COMMADAPTER>
</XML>
```

Here is an example of the defaults file for the Dehydrator.

```
<XML>
   <TimeOut MAX='5000' MIN='500'>1500</TimeOut>
   <HeartbeatRate MIN='250' MAX='30000'>1000</HeartbeatRate>
   <FW_Version>Version 1.3A</FW_Version>
   <RESPONSIVE>1</RESPONSIVE>
   <MODE>1</MODE>
   <PRESSURE MIN='0.0' MAX='999.9'>12.3</PRESSURE>
   <TEMPERATURE MIN='0.0' MAX='999.9'>23.0</TEMPERATURE>
   <DUTY_CYCLE MIN='0.0' MAX='999.9'>105.0</DUTY_CYCLE>
   <FLOW_RATE MIN='0.0' MAX='999.9'>8.0</FLOW_RATE>
   <ADDRESS MIN='0' MAX='80'>0</ADDRESS>
   <lowpressure>0</lowpressure>
   <highpressure>0</highpressure>
   <leakysystem>0</leakysystem>
   <hightemp>0</hightemp>
   <lowtemp>0</lowtemp>
   <lowvoltage>0</lowvoltage>
   <AU1heat>0</AU1heat>
   <AU1cool>0</AU1cool>
   <AU2heat>0</AU2heat>
   <AU2cool>0</AU2cool>
   <leakyshutdown>0</leakyshutdown>
   <dewpoint>0</dewpoint>
   <changecompressor>0</changecompressor>
</XML>
```

If the new Jelly Bean requires a user interface then an XML formatted screens file must be created. This file describes the content, look, and feel of the user interface for the Jelly Bean. This data driven approach of defining the user interface automatically extends the interface when the new Jelly Bean is added to the system. Here is an example of the screens file for the dehydrator.

```
<XML>
   <SCREEN>
      <WINDOW name='Primary' title='##: %GUINAME%' x='100' y='100' cx='550' cy='580'>
         <PANEL x='10' y='10' cx='260' cy='260'>
            <LABEL title='Unit Mode:' x='10' y='10' cx='120' cy='25'/>
            <CONTROL target='##' name='SCI_MODE' data='SCI_MODE' control='SChooser' active='true' x='140' y='10' cx='100' cy='25'/>
            <LABEL title='Unit Role:' x='10' y='40' cx='120' cy='25'/>
            <CONTROL target='##' name='SCI_ROLE' data='SCI_ROLE' control='SChooser' active='true' always='true' x='140' y='40' cx='100' cy='25'/>
            <LABEL title='Unit State:' x='10' y='70' cx='120' cy='25'/>
            <CONTROL target='##' name='SCI_STATE' data='SCI_STATE' control='SChooser' readonly='true' x='140' y='70' cx='100' cy='25'/>
            <LABEL title='Unit Fault Status:' x='10' y='100' cx='120' cy='25'/>
            <CONTROL target='##' name='SCI_FAULT' data='SCI_FAULT' control='SAnnunciator' readonly='true' x='140' y='100' cx='100' cy='25'/>
            <LABEL title='Firmware Version:' x='10' y='130' cx='120' cy='25'/>
            <CONTROL target='##' name='FW_Version' data='FW_Version' control='SEditField' readonly='true' x='140' y='130' cx='100' cy='25'/>
            <LABEL title='Data Refresh (ms):' x='10' y='160' cx='120' cy='25'/>
            <CONTROL target='##' name='HeartbeatRate' data='HeartbeatRate' control='SEditField' x='140' y='160' cx='100' cy='25' type='I'/>
            <LABEL title='Remote Timeout:'                                                      x='010' y='190' cx='120' cy='25'/>
            <CONTROL Target='##' name='TimeOut' data='TimeOut' control='SEditField' level='2'   x='140' y='190' cx='100' cy='25' type='I'/>
            <LABEL title='Last Update:' x='10' y='220' cx='0' cy='0' />
            <CONTROL Target='##' name='TelemetryUpdateTime' data='TelemetryUpdateTime' control='SEditField' x='90' y='220' cx='150' cy='25' readonly='true' />
         </PANEL>
         <PANEL x='10' y='280' cx='260' cy='170'>
            <LABEL title='Telemetry:'          x='65' y='10' cx='130' cy='25'/>
            <LABEL title='Pressure:'                                                            x='10' y='40' cx='90' cy='25'/>
            <CONTROL target='##' name='PRESSURE' data='PRESSURE' control='SEditField'           x='90' y='40' cx='70' cy='25' type='R' format='0.00' readonly='true'/>
            <CONTROL target='##' name='UNIT_PRESSURE' data='SI_UNITS' control='SChooser'        x='160' y='40' cx='90' cy='25' list='0!psi,1!mbars' readonly='true'/>
            <LABEL title='Temperature:'                                                         x='10' y='70' cx='90' cy='25'/>
            <CONTROL target='##' name='TEMPERATURE' data='TEMPERATURE' control='SEditField'     x='90' y='70' cx='70' cy='25' type='R' format='0.0' readonly='true'/>
            <CONTROL target='##' name='UNIT_TEMPERATURE' data='SI_UNITS' control='SChooser'     x='160' y='70' cx='90' cy='25' list='0!degrees F,1!degrees C' readonly='true'/>
            <LABEL title='Duty-Cycle:'                                                          x='10' y='100' cx='90' cy='25'/>
            <CONTROL target='##' name='DUTY_CYCLE' data='DUTY_CYCLE' control='SEditField'       x='90' y='100' cx='70' cy='25' type='R' format='0.0' readonly='true'/>
            <CONTROL target='##' name='UNIT_DUTY_CYCLE' data='UNIT_DUTY' control='SChooser'     x='160' y='100' cx='90' cy='25' list='%,%' readonly='true'/>
            <LABEL title='Flow Rate:'                                                           x='10' y='130' cx='0' cy='0' />
            <CONTROL Target='##' name='FLOW_RATE' data='FLOW_RATE' control='SEditField'         x='90' y='130' cx='70' cy='25' type='R' format='0.000' readonly='true'/>
            <CONTROL Target='##' name='UNIT_FLOW_RATE' data='SI_UNITS' control='SChooser'       x='160' y='130' cx='90' cy='25' list='0!cu.ft/min,1!liters/min' readonly='true'/>
         </PANEL>
         <PANEL x='10' y='460' cx='260' cy='110' bgcolor='lightgray'>
            <LABEL title='User Data:'          x='65' y='10' cx='130' cy='25'/>
            <LABEL title='Device Mode:'                                                          x='10' y='40' cx='130' cy='25'/>
            <CONTROL target='##' name='MODE' data='MODE' control='SChooser'                     x='140' y='40' cx='100' cy='25' active='true'/>
            <LABEL title='Device Address:'                                                      x='10' y='70' cx='0' cy='0' />
            <CONTROL Target='##' name='ADDRESS' data='ADDRESS' control='SEditField'             x='140' y='70' cx='100' cy='25'  />
         </PANEL>
         <PANEL x='280' y='10' cx='250' cy='440'>
            <LABEL title='Alarms:'                x='65' y='10' cx='130' cy='25'/>
            <LABEL title='Low Pressure'             x='10' y='40' cx='130' cy='25'/>
            <CONTROL target='##' name='lowpressure' data='lowpressure' control='SAnnunciator'        x='150' y='40' cx='75' cy='25'/>
            <LABEL title='High Pressure'      x='10' y='70' cx='130' cy='25'/>
            <CONTROL target='##' name='highpressure' data='highpressure' control='SAnnunciator'      x='150' y='70' cx='75' cy='25'/>
            <LABEL title='System Leak'              x='10' y='100' cx='130' cy='25'/>
            <CONTROL target='##' name='leakysystem' data='leakysystem' control='SAnnunciator'        x='150' y='100' cx='75' cy='25'/>
            <LABEL title='Shutdown:System Leak'    x='10' y='130' cx='130' cy='25'/>
            <CONTROL target='##' name='leakyshutdown' data='leakyshutdown' control='SAnnunciator'    x='150' y='130' cx='75' cy='25'/>
            <LABEL title='Low Temerature' x='10' y='160' cx='130' cy='25'/>
            <CONTROL target='##' name='lowtemp' data='lowtemp' control='SAnnunciator'                x='150' y='160' cx='75' cy='25'/>
            <LABEL title='High Temerature'          x='10' y='190' cx='130' cy='25'/>
            <CONTROL target='##' name='hightemp' data='hightemp' control='SAnnunciator'              x='150' y='190' cx='75' cy='25'/>
            <LABEL title='Low Voltage'         x='10' y='220' cx='130' cy='25'/>
            <CONTROL target='##' name='lowvoltage' data='lowvoltage' control='SAnnunciator'          x='150' y='220' cx='75' cy='25'/>
            <LABEL title='Unit 1 Will Not Heat'                                                      x='10' y='250' cx='130' cy='25'/>
            <CONTROL target='##' name='AU1heat' data='AU1heat' control='SAnnunciator'                x='150' y='250' cx='75' cy='25'/>
            <LABEL title='Unit 1 Will Not Cool'                                                     x='10' y='280' cx='130' cy='25'/>
            <CONTROL Target='##' name='AU1cool' data='AU1cool' control='SAnnunciator'                x='150' y='280' cx='75' cy='25'/>
            <LABEL title='Unit 2 Will Not Heat'                                                      x='10' y='310' cx='130' cy='25'/>
            <CONTROL target='##' name='AU2heat' data='AU2heat' control='SAnnunciator'                x='150' y='310' cx='75' cy='25'/>
            <LABEL title='Unit 2 Will Not Cool'                                                     x='10' y='340' cx='130' cy='25'/>
            <CONTROL Target='##' name='AU2cool' data='AU2cool' control='SAnnunciator'                x='150' y='340' cx='75' cy='25'/>
            <LABEL title='Dewpoint'                                                                  x='10' y='370' cx='130' cy='25'/>
            <CONTROL target='##' name='dewpoint' data='dewpoint' control='SAnnunciator'              x='150' y='370' cx='75' cy='25'/>
            <LABEL title='Change Compressor'                                                        x='10' y='400' cx='130' cy='25'/>
            <CONTROL Target='##' name='changecompressor' data='changecompressor' control='SAnnunciator' x='150' y='400' cx='75' cy='25'/>
         </PANEL>
         <CONTROL target='##' name='Reload' data='RELOAD' text='Reload' control='SPushButton'          x='0' y='0' cx='10' cy='10' bgcolor='green'/>
         <BUTTON title='Apply'                                                                       x='365' y='470' cx='75' cy='25'/>
         <CONTROL Target='##' name='SaveDefaults' data='SaveDefaults' text='Save' control='SPushButton' x='365' y='510' cx='75' cy='25' />
         <CONTROL Target="##" name="SCI_SYNC" data="SCI_SYNC" text="Sync" control="SPushButton"       x="280" y="470" cx="75" cy="25" />
         <CONTROL Target="##" name="RESET" data="SCI_RESET" text="Reset" control="SPushButton"        x="280" y="510" cx="75" cy="25" />
          <BUTTON title='Close'                                                                      x='450' y='470' cx='75' cy='25'/>
      </WINDOW>
   </SCREEN>
</XML>
```

If the new Jelly Bean provides a capability not found in any of the pre-existing Jelly Beans then a Virtual Instrument Jelly Bean may also be required along with its XML files.

Assuming the correct Jelly Bean already exists, it is selected from the bowl and added to the system. Adding it to the system only entails adding it to the Stovokor XML file and providing the executable, default, screen, and system setup files for the Jelly Bean (as exemplified above).

7 Projected Cost of Developing the New Architecture

This section will examine the cost that are projected to be spent in order to field the new architecture for Geo-synchronous Earth Orbit (GEO) satellites and Low Earth Orbit (LEO) satellites.

7.1 Geo-synchronous Earth Orbit (GEO) Satellites

As an initial starting point for implementing the new software, development of a GEO system was undertaken in the winter of 2000 and 2001. This course of action was chosen as the starting point because it was considered a subset of a LEO system and would therefore reduce the risk associated with development of a much larger, more complex, and more costly system. It was felt that all aspects of the architecture could be exploited, demonstrated, and rung-out with this initial development effort. Additionally, a customer was interested in utilizing the new architecture and was willing to provide a substantial amount of the funding required for the initial development effort. The remaining funding would be from the company's research and development R&D resources.

It was projected that a GEO system capable of performing the customer's requirements would require 7,143 man-hours of effort to develop. Of these hours, 3,333 would be funded with R&D funds, and the customer would fund 3,810.

Assuming a full time employee cost $162 / hour this initial development will cost $539,946 of R&D funds, and $617,220 in customer funds for a total of $1,157,166.

If the cost of Rhapsody is included the initial GEO develop cost is $1,157,166 + $228,000 = $1,385,166

7.2 Low Earth Orbit (LEO) Satellites

A simple LEO system has more capability than a GEO system. The antenna has a pedestal that allows it to point to any place in its celestial hemisphere. Also, it generally receives and transmits on one or two frequencies. It can perform additional system characterization tests. Additionally, it is expected to track multiple satellites during the course of a single day and therefore it must be able to ingest a more complex and demanding schedule. A complex LEO system can have a handle of receive and transmit frequencies, and generally includes some type of data recorders or data processing and switching equipment.

To develop a simple LEO system from the previously developed GEO system is projected to take an additional 1,500 man-hours of effort.

Assuming a full time employee cost $162 / hour this additional development will cost $243,000. If the GEO product is never deployed but is considered as part of the development cost of the LEO system, the total cost of developing the LEO system will be $1,400,166.

If the cost of Rhapsody is included the initial LEO develop cost is $1,400,166 + $228,000 = $1,628,166

8 Management Points of Interest

This section discusses the cost effectiveness of the new system. It will examine some of the reasons the new system is expected to be more cost effective. It will examine the cost of deploying the old architecture and the new architecture. And finally, it will examine how many systems must be sold to recoup the cost associated with developing the new system.

8.1 Factors contributing to the Cost Effectiveness of the New Architecture

Some of the factors influencing the cost of developing and maintaining the new architecture during its life cycle are tangible and intangible.

One factor is the projected reduction in modifications to deploy a new system and the ease of maintenance to a fielded system. The new system has been design to reduce changes to already developed source code. This was realized by utilizing XML files for default configurations and user interface screen definitions, and making the system data driven when ever possible as opposed to algorithm driven (hard-coded).

Another factor is the ability to test the software at the factory there by decreasing the dependency to finish development and testing of the software in the field. This has been realized by requiring each physical device in the system to be simulated by software that can stand-in for the real hardware during software testing.

Finally, development of the new system is of intellectual interest and therefore job satisfaction to the software personnel already with the company and those that are being hired. The old architecture is becoming technically obsolete relative to the software language and software inter-process communication changes that have occurred within the software industry since the time of its conception. The software engineers working on the old architecture are aware of this temporal and technical obsolescence and do not want there areas of knowledge and skill to become equally obsolete. The new architecture will stimulate the intellectual interest and maintain the trendy skill sets of the engineers that maintain the system software and also carry the corporation's intellectual knowledge. Although the cost of replacing these knowledgeable and capable engineers has not been quantified, it is a consideration in making the decision to develop the new architecture.

8.2 Cost Effectiveness of New Architecture

Once the new architecture is in place (i.e., the initial GEO and simple LEO systems are deployable) other systems can be built using them as a baseline or starting point.

The old architecture requires approximately 400 man-hours of effort to produce a new system that is "just like an existing system but with minor modifications". This effort includes changes to the overall system, device control, configuration management, user interface, and report generation. For systems that are very different and therefore requiring extensive changes the man hour effort can run into the thousands of hours, as demonstrated when approximately 7,500 hours were required to upgrade NASA's McMurdo Antarctica station.

Assuming a full time employee cost $162 / hour and knowing a simple LEO system with the old architecture requires 480 hours of effort to change, a cost of $77,760 will be realized if the old architecture is modified and deployed.

Using the new architecture for a simple LEO system it is projected the effort will be reduced to 80 hours for Jelly Bean changes and/or development and 1 hour for XML report file changes and/or development.

Assuming a full time employee cost $162 / hour and projecting the new architecture for a simple LEO system requires 81 hours of effort to change, a cost of $13,122 will be realized if the new architecture is modified and deployed.

Comparing the costs associated with creating a new system based on the old architecture and the new architecture shows that by utilizing the new architecture a savings of $64,638 can be realized for each system deployed.

8.3 Payback schedule

Is development of the new architecture cost effective? This section will attempt to answer that question.

8.3.1 GEO system

How many GEO systems must be deployed to realize enough savings to recoup the cost of development? To answer this question we must use a few projections and make a few assumptions.

- This analysis will not consider the time value of money or taxes.
- The cost of intangibles has not been considered.
- We sell an average of 6 systems each year.
- The GEO development cost is projected at $1,385,166.
- The effort and cost of modifying and deploying a GEO system using the old architecture is 480 hours at a cost of $77,760.
- The effort and cost of modifying and deploying a GEO system using the new architecture is 81 hours at a cost of $13,122.
- The system will be sold for the same amount of money regardless of which architecture is used. Therefore, the sale price must cover the cost of using the old architecture which causes the old sale price to be maintained. This equates to an expense savings of $77,760 - $13,122 = $64,638 for each deployment (as determined previously in this document).

If we consider the initial development cost of the new GEO system (including the 1st year cost of Rhapsody) as a negative and the deployment saving per system as a positive factor the new development is paid for after 22 systems have been fielded over a period of 3.67 years if we ignore the time value of money.

n = $1,385,166 development cost / $64,638 deployment savings per system.
n = 22 systems

years = 22 systems / 6 per year
years = 3.67

If we consider the time value of money, this chart shows the number of years to recover the cost of development for a desired level of IROR and if 6 systems are sold each year.

Year	IROR	Cash Flow	Cash Flow Description
0		($1,385,166)	GEO Development cost
1		$387,828	Savings realized during modifications of new arch. vs. old arch. for 6 systems sold per year
2	-33%	$357,828	
3	-11%	$357,828	
4	2%	$357,828	
5	10%	$357,828	
6	15%	$357,828	
7	18%	$357,828	

8.3.2 LEO System

How many LEO systems must be deployed to realize enough savings to recoup the cost of development? Using the same projections and assumptions that were used for a GEO system yields:

If we consider the initial development cost of the new LEO system (including the 1st year cost of Rhapsody) as a negative and the deployment saving per system as a positive factor the new development is paid for after 26 systems have been fielded over a period of 4.3 years if we ignore the time value of money.

n = $1,628,160 development cost / $64,638 deployment savings per system.
n = 26 systems

years = 26 systems / 6 per year
years = 4.3

If we consider the time value of money, this chart shows the number of years to recover the cost of development for a desired level of IROR and if 6 systems are sold each year.

Year	IROR	Cash Flow	Cash Flow Description
0		($1,628,160)	LEO Development cost
1		$387,828	Savings realized during modifications of new arch. vs. old arch. for 6 systems sold per year
2	#NUM!	$357,828	
3	-17%	$357,828	
4	-4%	$357,828	
5	4%	$357,828	
6	9%	$357,828	
7	13%	$357,828	

9 Conclusion

This section will briefly describe the main points discussed in the paper and draw a conclusion about the decision to design and implement the new architecture.

The old architecture is based on aging technology. This technology cannot take full advantage of distributed computing and network devices. It is complex to maintain and modify because of its global memory, static data structures, and large all encompassing programs. This causes most of the system's programs to be affected when a change is required.

The new architecture is based on the latest technology. This technology takes full advantage of distributed computing and can support network devices. Its modular design utilizes information hiding techniques allowing changes to occur to a system component (Jelly Bean) without affecting other components. The modular nature of the design allows new systems to be assembled from the existing set of components without the need for major software changes. The new technology is trendy and progressive. This will allow the company to market itself as cutting edge and to attract talented engineers to work on the project.

The new architecture will reduce the time to market for a specific system there by making the company more responsive to a customer's schedule and deadlines. This will allow for greater internal scheduling and resource allocation across all projects that are on-going within the company.

The new architecture's reduced cost of deployment will give the company greater flexibility to negotiate price and allows for a greater reduction in selling price. The ability to lower prices and still make a profit may make the difference in realizing or losing a sale.

It appears to be a sound decision to develop and implement the new architecture for both tangible and intangible reasons. The technologies promote the company as mainstream and progressive and will attract customers and engineering talent. The projected payback schedule shows the systems will be paid for in the early years of their expected life. And finally, the new architecture provides greater flexibility to gain market share by reducing the time to market and providing more pricing strategies.

10 Appendix

10.1 Evaluating Rhapsody

A tool such as Rhapsody can save a project time and effort provided it is the correct tool. If it is not the correct tool the organization can:

- Waste a lot of money and schedule time
- Be disrupted from developer frustration
- Find itself with a developer morale problem

The following approach could be used to evaluate a tool such as Rhapsody before the organization makes a complete commitment:

- Define the minimum set of features the tool must perform. Determine how each of these features will benefit the organization and what value the feature will bring to the development effort.
- Perform an industry wide search for all tools that perform the same functionality.
- Evaluate the tools against the feature criteria.
- If a feature is not available in a particular tool, determine if it is critical to realizing the benefits of purchasing the tool.
- Once the best tool that passes the minimum criteria is identified, establish a proof-of-concept project to test the tool in a pseudo real development environment.
- Obtain the funding and the time for appropriately skilled personnel to evaluate the tool by using it for a proof-of -concept project.
- Establish the minimum criteria that must be achieved during the proof-of-concept project. This should include such features as efficiency increase in the members ability to produce more software in a shorter time, time to master the tool, ease of use of the tool, and level of expertise required to drive the tool.
- Evaluate the tool against the proof-of-concept criteria.

10.2 Learning Curve Costs

A tool such as Rhapsody has a learning curve. This learning curve will cost the project time and money. The time spent overcoming the learning curve should be considered when determining the true cost of acquiring a new tool. Knowing what the learning curve cost will be is not an easy question to answer. The following criteria could be considered to get an understanding of this cost:

- Metrics collected during the evaluation of a similar tool.
- Collect metrics during the evaluation phase of the current tool during the proof-of-concept evaluation period.
- Ask the vendor for metrics.
- Acquire metrics from other customers that have purchased and evaluated the tool.

11 References

[Henning 1999] Henning Michi, Vinoski Steve, 1999, Advanced CORBA Programming with C++, Massachusetts, Addison-Wesley

[Douglas 1999] Douglas Bruce Powell, 1999, Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns, Addison-Wesley Longman, Inc.

[Lemay 1999] Lemay Laura, Cadenhead Rogers, 1999, SAMS Teach Yourself Java 2 in 21 Days, Sams Publishing

[Manison 2001] Manison Andrew, 2001, Stovokor Architecture Description, ViaSat Inc., 2001.

[Park 2002] Park Chan S., 2002, Contemporary Engineering Economics, Third Edition, New Jersey, Prentice Hall

[Rhapsody 2001] http://www.ilogix.com/products/rhapsody/rhap_incplus.cfm