DESIGN OF AN OBJECT ORIENTED LANGUAGE SYNTAX FOR UIL, THE USER INTERFACE LANGUAGE OF THE SPACE STATION FREEDOM

by

Garrett Tim Sos

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
MASTERS OF SCIENCE
With a Major in ELECTRICAL ENGINEERING
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 8 9

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Dr. François E. Cellier — November 29, 1989

Dr. François E. Cellier
Associate Professor

Date

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

ABSTRACT

The design of a new computer language, called the User Interface Language (UIL), is analyzed and evaluated by coding a representative procedure. UIL will provide the man-machine interface for command procedures on the Space Station FREEDOM. The UIL procedure written is modeled after an operational procedure used in the Space Shuttle program. This work provides a concrete test case to verify that UIL can be used to implement procedures for the Space Station. The object oriented approach taken with UIL is based on the successful application of these concepts for a variety of other software tools in operation today. Three major enhancements are proposed in this thesis: event handlers, data structures, and class/object creation capabilities. The addition of these capabilities changes the character of UIL from an object manipulation language to an object based language. The new capabilities, if adopted, will profoundly change the future of UIL.

1. INTRODUCTION

NASA, in conjunction with several international partners, is currently designing the most complex system ever to be deployed in Space, the Space Station FREEDOM. The hardware and software in this system will be many times more complex than any previous NASA effort, including the Space Shuttle program. The success of the effort will rely on many new innovations in hardware and software technologies. In this thesis, we will consider the design of a new software language, called the User Interface Language (UIL), for use on the Space Station project and reflect on the design of UIL by writing an example UIL procedure modeled after an operational control procedure used in the Shuttle program.

UIL will provide the man-machine interface for most of the test and operation command procedures on the forthcoming Space Station FREEDOM. UIL will be structured much like a natural language, and will be object oriented. UIL will also be extensively used for the development and testing of experiments on the ground, and the remote execution of these experiments. UIL, for example, could be used to command, from the ground, hardware valves and pumps located within a laboratory located aboard the Space Station. The command to open the valve on a heat exchanger might be: "OPEN VALVE OF HEAT EXCHANGER TO 85%".

We will implement an operational Ground Operations and Aerospace Language (GOAL) procedure, called SSLO2, from the Space Shuttle launch system in UIL and comment on the ability to use UIL for the operations performed by SSLO2. We shall accomplish this by using a computer program called a parser that allows us to implement the syntactical description of the UIL language. The parser allows us to write the SSLO2 example in UIL code and verify that the UIL version of the SSLO2 is syntactically correct. The UIL version of SSLO2 then provides us with the information needed to verify that the language can be used to reasonably implement SSLO2. We will show that, to completely implement the SSLO2 example in UIL, several extensions must be added to the UIL syntax.

This document is organized in the following manner: The remainder of this chapter discusses the background and requirements for UIL in the Space program. Chapter 2 discusses the general concept of object based systems, the rational for using an object based approach in UIL, and general high level goals for the design of UIL. Chapter 3 discusses, in detail, the proposed syntax of UIL. Chapter 4 develops in detail the SSLO2 example. Chapter 5 discusses each proposed extension of the UIL syntax needed to implement SSLO2 and provides a rationale for the proposed extensions. Finally, Chapter 6 will review the observations made about UIL and provide a summary of the conclusions. There are also several appendices that contain useful information including: a listing of the UIL syntax definition, a short GOAL code example, the SSLO2 procedure written in UIL and a list of acronyms.

1.1. The Role of UIL

UIL is being designed to provide users powerful access to Space Station resources without the need to write procedures in a conventional computer programming language, such as ADA. UIL will be used by a wide range of personnel including scientists, test and operations personnel, and astronauts. Most of these users will not be trained computer programmers or engineers and cannot be expected to write ADA procedures to perform their day to day jobs. Conventional operating system command shells, such as VMS DCL, DOS *.bat files, or Unix shell scripts, can provide some of the functionality needed by these users; however, these command shells are not sufficiently powerful for extensive use in the Space Station environment. UIL supports specific functions that allow access to Space Station resources and also provides a command language that is consistent across all of the operating systems that will be used in the Space Station environment. UIL will fill the gap between the operating system and conventional computer programming languages.

Many of the concepts of object based systems are included in UIL to provide the users with a more familiar data abstraction for hardware devices then found in conventional computer programming languages. In UIL, most Space Station subsystems will be represented by a computer data type called an object. This computer 'object' is the internal computer version of a physical device on the Space Station, for example, a valve. Commands that are issued to the computer object will, in turn, cause the computer to direct the physical object to change state. This command to an object is called an 'action' in UIL. It is also possible for an object to have internal components that are also objects. In the example "OPEN VALVE OF HEAT EXCHANGER TO 85%", the "HEAT EXCHANGER" is the object that represents a complex system, the "VALVE OF" is an internal component of the heat exchanger and is also an object. The "OPEN ... TO 85%" is the action that will be applied to the object "VALVE" within the "HEAT EXCHANGER" subsystem.

Any changes of state within the Space Station will be specified by an object-action pair. An object can be tested, stopped, started, examined, or asked to perform any other actions that it understands how to respond to. Each object may behave differently, based on the type of object it is, to the same action. Objects will only respond to the actions that have meaning for them. The object itself will know the correct way to respond to a given action. For example, the action "OPEN" would perform a much different set of procedures for an electric valve than it would for a pneumatic valve. Both types of valve objects would receive the same action, that is to "OPEN", and would execute the correct code required for each valve type. The burden of issuing the correct low level commands to the hardware becomes the object software responsibility and not the responsibility of the user of UIL.

UIL, and the many other tools that will be built to support it, will contain elements of data base managers, display systems, user input/output interfaces, and remote control languages. UIL will supplement and replace development in earlier aerospace languages, such as GOAL. UIL will provide an easy to understand method of describing and controlling complex hardware subsystems. The objects will also contain additional procedures to support the simulation of the object behavior. This will allow debugging software procedures before the hardware is complete, or to support operational failure diagnosis.

It is expected that the Space Station will be composed of many thousands of software 'objects' representing subsystems built by numerous contractors and international partners. UIL will be used in all ground based testing and operations, new launch subsystems, and operation procedures on the Space Station. The scope of the challenge is expected to be immense: There could be more that 20,000 object/action pairs in the Space Station. UIL will also define the model that will be used to represent data and control for all objects within the Space Station, and defines the structure that NASA must use to manage the development and configuration of Space Station subsystems and ground support equipment (GSE).

1.2. Syntactical Descriptions of Computer Languages

A computer language is a high level description of commands and procedures that is written to perform a particular function. These high level commands are turned into low level instructions that can be understood by the computer. A computer language consists of both a syntax and of semantics. The syntax corresponds to the structure of the statements in the language, whereas the semantics describe the meaning of the statements. The semantics of a new language is contained in the written description of the function that each statement performs.

In order to evaluate the usefulness of a new computer language to an application, there must be a formal description of the language. The formal description for the syntax of UIL used here will be presented in the Backus-Naur Form (BNF). Given a BNF form of a new language, it is possible to verify the syntactic correctness of a procedure written in the language in an automated way by using a syntactical analysis tool called a parser. The language syntax can only be used to verify that these rules are followed to form a syntactically correct program, it cannot determine if the program is logically or semantically correct. In the evaluation of UIL, a syntax parser procedure will be used (hereafter called simply the parser).

With the parser procedure, the syntax description of UIL is input along with example procedures written in UIL. The parser will output information on the correctness of the syntax of the example procedure. Semantic correctness will be verified manually by analysis and review of the procedure. The parser used by the author was written by A.

P. Bongulielmi[1] and is based on an algorithm by N. Wirth[2]. The parser was written in Pascal and has been ported to many different machines, most recently to an Apple Macintosh II by the author. With this parser, we will be able to test the syntax of UIL procedures before the language is implemented.

1.3. Formal Requirements for the UIL

The discussion of UIL in this thesis is based on the specification of UIL[3] dated 12 July 1989 entitled *'Specification for the Space Station UIL Version 1.0'*. The writing of the UIL specification was a group effort that included Francois Cellier, University of Arizona, Randy Davis, University of Colorado, Cliford Grim, IBM, Charles Shaw, Century Computing, the author, and many others. Randy Davis deserves special credit as he was responsible for writing the majority of the 150 page long UIL specification. There is significant overlap between portions of this thesis and the UIL specification, most notably in Chapter 3. The UIL specification provides more detail in the specification of the semantics of the UIL; therefore, the reader interested in additional detail is encouraged to also review the UIL specification.

UIL has evolved from many other languages used in the Space program, from advances in user interfaces, and from object based languages. Figure 1.1 shows many of the languages that UIL has evolved from.

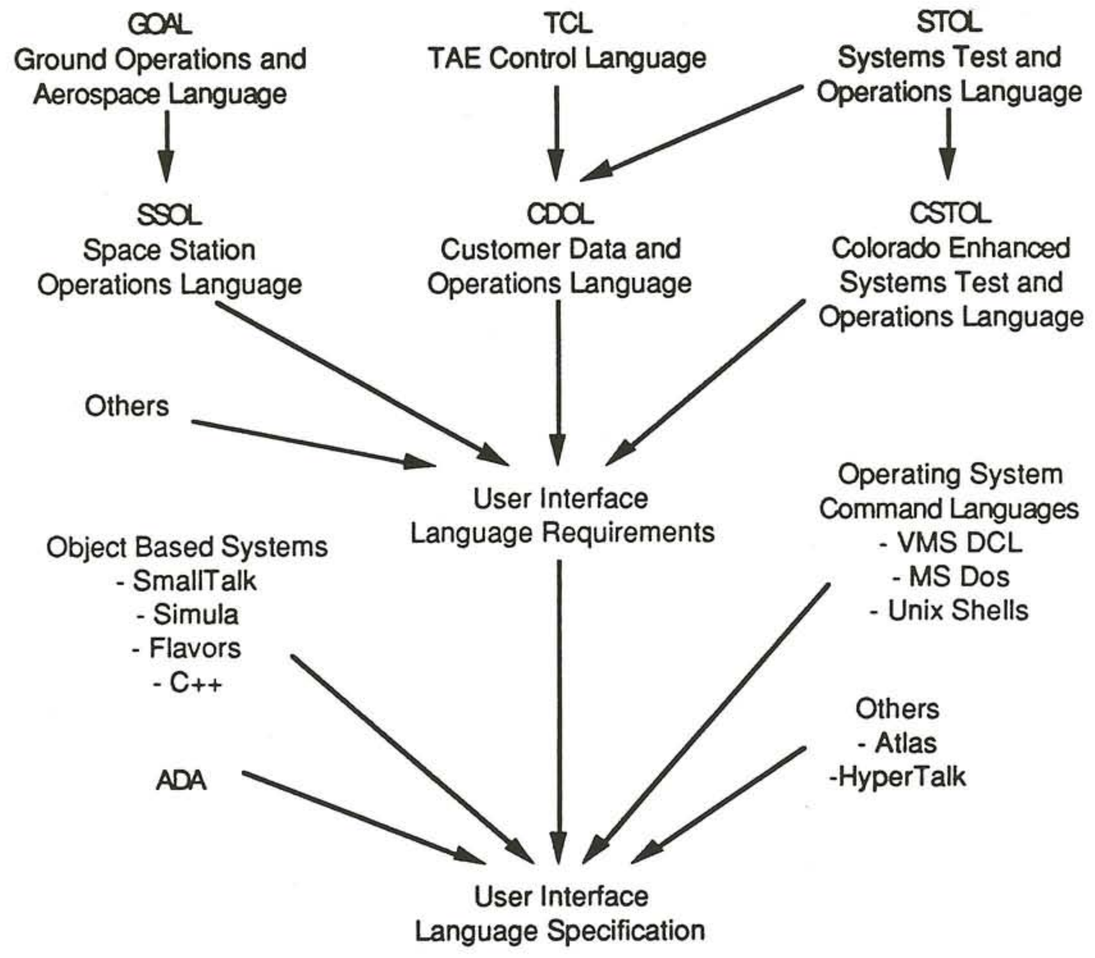

GOAL
Ground Operations and Aerospace Language
TCL
TAE Control Language
STOL
Systems Test and Operations Language
SSOL
Space Station Operations Language
CDOL
Customer Data and Operations Language
CSTOL
Colorado Enhanced Systems Test and Operations Language
Others
User Interface Language Requirements
Operating System Command Languages
- VMS DCL
- MS Dos
- Unix Shells
Object Based Systems
- SmallTalk
- Simula
- Flavors
- C++
Others
- Atlas
-HyperTalk
ADA
User Interface Language Specification

Figure 1.1 - The UIL Family Tree

Two important ancestors for UIL are GOAL[4], developed at Kennedy Space Center and used for the Shuttle test and launch operations, and STOL[5], developed by Goddard Space Flight Center and used in the test and operations of many Earth-orbiting spacecraft. Both of these languages were developed by NASA. They and another language called TCL[6] were found not to completely meet the requirements for UIL. Three newer languages, SSOL[7], CDOL[8] and CSTOL[9] were also found not to completely meet the needs for the Space Station project. None of these languages would produce the modern, easy-to-use and easy-to-maintain language required for the Space Station. It was also felt that several new and innovative programming concepts were needed within UIL to support the software programming tasks that will be undertaken for the Space Station. Therefore, the group working on the requirements formed an updated statement of requirements for UIL contained within the *'User Support Environment Functional Requirements Document'*[10]. Concepts from object based languages have also contributed extensively to the design of UIL (discussed in Section 2.2). A user interface prototype[11] also provided input to the design of UIL. Finally, UIL was designed to be compatible with ADA[12], as ADA is the language that will be used for all conventional programming on the Space Station project. It is also expected that the actual implementation of the data structures and actions of most objects used by UIL procedures will be written in ADA.

2. UIL DESIGN ISSUES

This chapter introduces the reader to object based systems and discusses several of the issues that influence the design of UIL. The rationale for the object based design of UIL is also given. The chapter is motivated by the author's experiences in porting a large GOAL procedure called SSLO2 into UIL. SSLO2 is a procedure from the operational software currently used to launch the Space Shuttle, and should be typical of the type of programming that UIL will be used for on the Space Station. Chapter 4 discusses the translation of SSLO2 into UIL in more detail.

Porting SSLO2 to UIL allows testing aspects of the design of UIL better than might have been done by simply 'designing' a language on paper. Indeed, the experience brought to light several issues that the current design of UIL does not address. While the debate continues as to the exact nature of UIL, this thesis addresses one design area in depth, namely whether UIL can be used to replace GOAL for writing procedures similar to the ones that are currently used on the Space Shuttle program. The answer is a qualified yes.

In this chapter, we intend to provide some rationale for the object orientation of UIL and outline a 'high level' set of goals that UIL must meet to replace GOAL as a development language. These high level goals can be viewed from the point of view of a manager of a group of GOAL code developers on the Shuttle program. This manager would ask questions such as: Are the basic concepts used in UIL proven to be reliable and useful?, Does the design of UIL support effective solutions for the software that I currently develop in GOAL?, Will UIL simplify the development of my software?

The answers to these questions are, to a great extent, found by examining the 'new concept' in aerospace languages that UIL introduces: object based design. There are three factors about object based design that should convince the manager that this 'new concept' will permit UIL to meet his goals for the language. These factors are as follows: the object data paradigm is useful for the representation of the Space Station subsystems, object oriented systems have been used successfully in recent years to simplify many designs. Object oriented programming should also increase the reliability and reduce the cost of UIL programming. Let's discuss the details of an example 'object' and address these issues as the object based concept is developed.

2.1. An Example Object

An object is a version of a physical device in the computers memory. The valve shown in Figure 2.1 is almost as simple as one which you might find in an automatic plant watering system. The valve has an input port and an output port that are connected to other physical devices, for example, the main water line and some sprinklers (we shall later represent these other devices as additional objects in the computer's memory). The control input is attached to an electric solenoid that opens the valve when power is applied. For the Space Station, the valve cannot be quite as simple because the valve may be part of some mission critical or life sustaining subsystem. It is no longer sufficient to command the valve to open and just hope it does! To ensure that we really know that the valve has worked correctly, let's add two sensors to the valve: one that will indicate when the valve is fully open and one to indicate when the valve is fully closed (called status-open and status-closed). Although this seems to be a simple change, we shall see in a later chapter just how difficult these two sensors can make the control procedures for this valve.

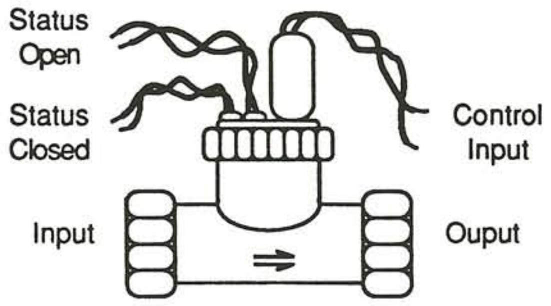


Figure 2.1 - Simple Electric Valve

In the computer memory, this valve can most simply be represented by three procedure calls as shown in Table 2.1. There is no data associated with the valve as there is nothing that the computer needs to remember about the valve. Two procedure calls read the open and close status from the valve support hardware ports directly, and one procedure call writes the command to open or close the valve to a hardware port that activates the solenoid. The programmer would execute any one of the three procedure calls when ever needed - this is an example of standard procedural programming. This type of programming is supported by most languages that are in use today (C, Fortran, Pascal, ADA, ...). These three procedures might be placed in a library or a special file and can be carefully controlled or 'packaged'; however, they are still simply procedure calls. If another valve was to be added to the program, three more routines would have to be added to the library with different port numbers.

Table 2.1 - Simple Valve Data Structure and Procedures

Device name: Very Simple Valve	
Data:	***Procedures:***
n/a	Read Open Status from input port 23
	Read Closed Status from input port 24
	Command Value at output port 25

A clever programmer would notice that several valves may need to be controlled, and add additional computer data memory to remember, or store, the values of the input and output port numbers for each valve as shown in Table 2.2. As valves are added, the three routines would be used to perform the desired functions on each new valve. The programmer now has an additional problem, where to store the information about the port numbers of each valve. This, of course, is simply solved by using arrays or data structures. However, as subsystems become more complex, the programmer is faced with a new problem: new devices to support (such as pumps, tanks, and motors). These new devices can be supported by adding new types of data structures. The more device types you have, the more complicated the control and memory management will become.

Table 2.2 - Valve Data Structure and Procedures

Device name: Valve	
Data:	***Procedures:***
open-input-port = port 23	Read Open Status from open-input-port
close-input-port = port 24	Read Closed Status from close-input-port
command-port = port 25	Command Value at command-port

Conventional procedural languages, including ADA, are of limited help to the programmer in his task of supporting the growing number and complexity of devices and their interconnections. The basic problem is that the conventional procedural languages 'bind' the execution order of procedures at compile time, that is to say, when the compiler generates code, the execution of a procedure is fixed into memory. If the programmer desires to execute one procedure if the device is a valve and another procedure if the device is a pump, then the programmer must test the device type and then call the desired routine. Every programmer is familiar with this type of programming, and clever programmers create very intricate and elegant ways of handling complexity. Object based support in a language can simplify these applications and provide a more consistent method of handling data and procedures.

At the most basic level, object based languages provide a data driven method of executing procedures (ADA supports this only to a limited extent with operator overloading, see Section 5.2 for more details). Table 2.3 shows the description of a simple object, called valve 1. Valve 1 has the same data and procedures as the last example, and something new called actions. The data and procedures of the last example (shown on the left side of the table) are now internal to the new object. The external interface to the object are the actions, shown on the right side of the table.

Table 2.3 - Valve 1 Object Structure

Object name: Valve 1	
Internal Object Data:	***External Actions:***
open-input-port	Close
close-input-port	Open
command-port	Read Status
Internal Object Procedures:	Set Ports
Read Open Status from open-input-port	Initialize
Read Closed Status from close-input-port	Print Status
Command Value at command-port	Help
Read Command Status at command-port	

Actions are procedures that are called by the object system function dispatcher based on the kind of object the action is sent to. You can think of the action as a message that is sent to an object. Once the object receives the message, the object interprets the action by executing the action procedure that produces the desired result. For example, if you send the action 'OPEN' to Valve 1, then the object system will execute the open procedure referred to in the declaration of the object Valve 1.

The action message causes the open procedure to execute in the same manner in which any other procedure would execute. What is different about the object orientation is that the actual call to the correct open procedure was determined by a dispatcher by examining the 'object' type that the action was sent to. Had you sent the action 'OPEN' to a 'DOOR', the dispatcher would have executed a different routine. The advantage is that the programmer does not write code that determines which open procedure needs to be called or how the open takes place. In the case of the Space Station, the programmer of UIL code will primarily manage objects and the actions to send to them, and will seldom worry about how to 'OPEN' a valve or how the object finds the correct open procedure for execution.

In Table 2.3 we added several other actions to provide the valve with a full set of support. The 'OPEN' and 'CLOSE' actions cause the valve to change state, the 'READ STATUS' action returns the valves position as best as can be determined, the 'SET PORTS' action sets the input and output port addresses, the 'INITIALIZE' action sets the objects internal data to a known state, the 'PRINT STATUS' action outputs the objects status to the operator, and the 'HELP' action returns information on the object, and on the data and actions that it contains.The low-level procedures still exist in this object, but they are now only used privately by the object's action procedures.

A more concise definition for UIL objects follows (a more detailed discussion of these definitions can be found in any of the references discussed in the next chapter):

(1) An OBJECT CLASS, or simply CLASS, is a formal description of the object structure including the private memory, the private procedures, and the set of action procedures it contains. In UIL, a class cannot receive actions, it can only be used to instantiate (bring into existence) a new object. In some object based implementations, the class itself is an object that can receive actions.

(2) An OBJECT, instance, or instantiation of an object is the data structure that contains the attributes and actions associated with an object, the attributes and actions as defined by the class that it was instantiated from. An attribute value refers to the value of the attribute in the object's memory.

(3) Instantiation is the act of creating an object of a particular class type.

(4) An ACTION is a request that is sent to an object to carry out an operation. Within the object, the action procedure is the description of how to perform the requested operation.

(5) A SUBCLASS is a new class (with a new name) created by adding additional private memory, private procedures and action procedures to an existing class. The new class is said to INHERIT all the features of the original class.

(6) A SUPERCLASS is the class from which a subclass has inherited features (see subclass).

(7) If a subclass is allowed to inherit features from more than one class, the object based system is said to allow multiple inheritance. The subclass contains all the features of all the classes that were used to create it.

(8) If an action procedure is defined in both the superclass and the new subclass, then the action procedure in the subclass will be executed, and it is said to override the action of the superclass. There is often provided a technique that allows the execution of the action procedure of the superclass from within the subclass action procedure.

(9) Binding is the term used to describe the time when the action procedure is fixed into code. The binding can happen 'early', i.e. at compile time (executes faster, is less flexible), or 'later', i.e. at execution time (executes slower, is more flexible).

Using an object based design is a convenient method of organizing data and procedures in a compact way. The object representation can be applied to many application problems with positive results. The object representation should be thought of as a tool for the programmer for solving data management problems. Objects should not be overused, and they do not replace conventional data structures for many applications. The process of dispatching actions against objects consumes computer time that may not be justified or available. The dispatching overhead is most evident in applications that are characterized by a large number of very small objects.

2.2. Object Oriented Systems

There are many examples of successful object oriented systems and languages in use today. Several current examples of available object based languages are as follows (most are commercial products): C++[13], Objective C[14], Actors[15], Flavors[16], Loops in Lisp[17], Scoops in Scheme[18], Classic ADA[19], and Smalltalk-80[20]. The basic concepts of object orientation have been discussed and used for many years (see '*Structured Programming*' by Dahl, Dijkstra, and Hoare[21] or the simulation language called SIMULA[22] for some of the early implementations). The best known example of a full featured object based computer environment is Smalltalk-80.

One of the most successful applications of object based design is found in handling the user interface and window systems in computer workstations available today. Due to the easy implementation of object based concepts in Lisp, most versions of Lisp support some type of object structures, for example, Flavors, Scoops, or Loops. Consequently, object based window systems have become the standard for Lisp workstations. New graphic and user interface standards other than Lisp workstations are now being designed to be object oriented, for example, widget tool kits for X-windows systems[23]. The lack of object support in conventional languages has been one of the drawbacks preventing a more extensive use of object based design.In the case of X-windows, most of the widget tool kits have been built by using conventional C compilers. This requires the object orientation to be 'programmed' manually by using conventional features of the language. In recent years, object concepts are being retrofitted into conventional languages such as C (C++, Objective C, available object support libraries), and ADA (Classic ADA). The availability of these languages should allow more widespread use of object based design.

The trend toward the use of object oriented languages is also fueled by the hope that using an object based approach will reduce the cost of software development. Some of the rationale behind the development of Objective C, for example, was that object based languages increase the ability of software to be reused. The compact nature of object data and procedures makes it easier to hide the details about the object. This increases the portability of the objects and code. The success of object based systems provides the answer to the managers first question: are the basic concepts proven to be reliable and useful? The basic concepts of object oriented systems have been developed over the last 20 years. Modern workstations depend, in many cases, on object based concepts to provide a solution for programming the complex window systems and user interfaces. And finally, there are now a considerable number of languages in existence that support object based programming.

2.3. ADA compatibility

ADA will be used to build the attributes and actions for most objects for UIL execution. While ADA does support packages, data hiding, and operator overloading, it only provides limited compile time binding, and does not support inheritance (see Table 2.4). These features, if they are deemed necessary, can either be added to ADA, or an ADA compiler with object based extensions can be used. UIL is required to be compatible with ADA.

Table 2.4 - Object Features of Several Languages

Feature	Smalltalk	Ada	C++	Objective-C	UIL
Binding time	late	early	either	either	TBD
Overloading	yes	yes	yes	no	no
Inheritance	yes	no	yes	yes	TBD
Multiple inheritance	yes	no	no	possible	TBD
Commercial availability	yes	yes	yes	yes	no

As can be seen in Table 2.4, there are many technical issues concerning the implementation of UIL that have not currently been resolved. The exact implementation of UIL is not the subject of this thesis; however, we will show in later chapters that some of the implementation details cannot be completely separated from the design of the syntax of UIL.

2.4. Objects and Hardware Subsystems

As will be demonstrated in the following chapters, our experience with SSLO2 has shown that an object oriented approach will work well for the control of complex hardware such as the systems that will be used within the Space Station. No other programming paradigm appears to provide the association of data and procedures in the efficient way that an object based approach does. The following factors are considered most important:

(1) The object oriented paradigm matches the problem, that is, the object representation of software devices closely models that of physical reality. The UIL programmer will deal with objects in software at the level of valves and motors, not bits and bytes.

(2) In using UIL to describe hardware subsystems in the Space Station, there will be many instances of each object class. An object oriented system is used most efficiently in an environment where there is more than one instantiation of each object class.

(3) Class Inheritance can be used effectively for many hardware objects in the Space Station. Classes for hardware subsystems will group software objects into hardware groups (For example, pumps, valves, or pipes). Subclasses will represent the more specific types of a class (For example, electric valve, pneumatic valve, and big valve). Multiple inheritance can also be applied in a physically meaningful way, for example, the basic valve class can be combined with a liquid oxygen (LOX) class to make a new subclass of LOX Valve (cf. the SSLO2 example of Valve + LOX precautions).

The simple mapping of object based representation to hardware subsystems provides the answer to the manager's second question: does the design of UIL support effective solutions for the software I currently develop in GOAL? GOAL supported the representation hardware at only the most basic level, for example, at the level of input and output ports. Contrary to this, UIL provides a higher level data and procedure organization structure that will support effective and efficient solutions for programs now written in GOAL code.

2.5. Writing UIL

One of the most noticeable differences from GOAL development will be how the procedures are written. In GOAL, there was no effective way to decompose large control programs into smaller units. The object orientation of UIL will allow complex procedures to be written in an easier manageable and more economic fashion. In UIL, most of the programming will involve the creation of robust computer objects used as interfaces to the control procedures for valves, pumps, and other hardware items (most of which will be coded in ADA). Once these objects are implemented, building subsystem models will be simple, and writing UIL programs that control or monitor these subsystems will be easier than writing equivalent GOAL procedures for the following reasons:

(1) An entire hardware subsystem object will be created by simply connecting together many smaller objects. The subsystem object will contain the information of the way in which the simpler objects are interconnected within the actual hardware subsystem.

(2) Hardware subsystem objects also contain control information by having access to the actions of the simpler objects (For example, the valve will know how to perform basic valve operations). Additionally, any voting strategy used to determine the actual state of an object in the subsystem will be built into the 'schematic' of the subsystem object. A voting strategy is a method used to determine the state of something based on several measurements, for example, if there are two votes for open and one vote for closed, then the valve is taken to be open.

(3) Hardware subsystem objects will contain current state information, both from a programmatic point of view (For example, what the program is trying to accomplish: opening a valve, starting a pump, ...) and from a physical point of view (For example, the measured temperatures, flow rates, switch settings, ...).

(4) Each object within the subsystem will also contain computer models of themselves to allow testing and simulation of the subsystem itself. This will allow off-line experimental simulations of the physical system in the event of unexpected operating conditions or for developmental testing(5). The subsystem will be able to communicate status information to the outside world (For example, to operators, printers, ...) without hard coded device names or paths.

(6) The objects should also contain the information needed to model component failure modes for both off-line simulations and for control of operational systems with actual component failures.

All the information about controlling, testing, or simulating a hardware subsystem can be contained in the object structure. This object structure will be consistent for each program that will be written (testing, simulation, controlling, ...). The additional information in the structure is not harmful as only the information needed for a particular task is used.

The clean representation of data, control, and interconnections of objects provides the answer to the managers third question: will UIL simplify the development of my software? When procedures were written in GOAL, there was no support for higher level data structures. Each programmer devised his own 'clever' solution, and all information had to be programmed into the code; therefore, there was little overall structure or consistency to the programs. UIL provides an intelligent, high level structure for all device specific information.

2.6. UIL Execution Issues

UIL represents a departure from a long tradition of aerospace languages. UIL is a modern, object oriented, modularity structured, hierarchically decomposable language. Most languages and programs previously used in aerospace were completely flat. Often, even subroutines were not used but replaced by sequences of 'goto' statements within an amorphous, and almost unstructured large piece of code. Reasons for this traditional approach to aerospace programming are the need for fast program execution, and the need to be able to measure the execution time of particular program sections accurately. However, on the other hand, programs written in this manner are very expensive to properly maintain, and it is difficult to prove their correctness. It is felt that, with the advent of faster computers, the need for time-optimal programming has become less imminent. Also, if the need arises for optimization of time-critical portions of the code, the UIL processor could easily be equipped with a compiler switch that would request a particular program section to be flattened out for improved run-time efficiency.

2.7. Object Manager

To provide operational management over objects that represent operating hardware, it is expected that a set of graphical and textual tools will be available to build, edit, and manage the 'operating' hardware subsystem objects. There are at least two uses of object representations of hardware subsystems: controlling actual hardware and running simulations of the hardware. There will be an object manager that will give one, and only one, operating procedure intrusive access to the actual hardware at any one time (call this the 'operational-copy' of the object). In the Space Station environment, the object manager is complex due to the fact that there may be two, or more, controllers for the same hardware (either due to redundancy of hardware or to allow greater flexibility). This object manager will allow as many 'simulation-only' copies of the object as requested by users needing them. Nonintrusive access to the hardware (e.g. reading the current valve position) may also be granted to several controllers simultaneously.

In all likelihood, the implementation of the object manager will operate somewhat like distributed database systems. For example, one method that might be used would support the concept of a 'operational-copy' token. The processor holding the token is the only processor that is granted intrusive access to the hardware. This token is initially passed from the object manager to the processor in charge of the hardware. The token, in turn, can be passed back to the object manager, or to another processor desiring future control of the hardware. There will also be support to allow the regeneration of a token that is lost due to a processor or a network failure.

3. UIL SYNTAX DESIGN

This chapter summarizes syntactical differences between UIL and conventional languages, such as C or ADA, and discusses several extensions added to UIL by the author. The version of UIL used in this thesis is a modified form of the version published by the User Support Environment Working Group (USEWG) dated 10 June 1989. The organization of this chapter parallels Chapter 3 of the USEWG document. The version of UIL used in this thesis differs from the original in two major respects: this version can be parsed with an LL(1) parser, and this version supports the creation of complex objects in UIL. LL(1) stands for left parsible with look ahead of one symbol only. Appendix A contains the complete listing of the UIL syntax. Only the significant differences between UIL and conventional languages will be discussed in this chapter. The reader who is interested in a more detailed description of UIL is directed to the USEWG document.

This thesis uses the LL(1) version of a language syntax. An LL(1) version is convenient for study because it is possible to obtain a parsing algorithm which determines the syntactical correctness of test program fragments. It does this by scanning the input file from left to right while looking ahead only one symbol. The algorithm is straightforward, and there exist tools to examine such a language syntax before the language is actually implemented. The parsing tool used in this thesis works with LL(1) grammars only. The original UIL grammar was modified to permit the use of the LL(1) parser. All features of the original UIL syntax were represented in the LL(1) version. The only inconvenience in the LL(1) form is that it is not as easily readable by humans; therefore, both the USEWG and the LL(1) versions are shown in the Appendix. The description of the syntax in this chapter will use the USEWG format.

The USEWG version of UIL does not allow procedures to define or create complex objects. For the task of porting SSLO2 into UIL, UIL must be able to define and use complex objects without the aid of objects built external to UIL. SSLO2 is a procedure from the operational software currently used to launch the Space Shuttle, and should be typical of the type of programming that UIL will be used for on the Space Station. The syntax that will be used to create complex UIL objects is provided in this chapter.

3.1. Notation

The syntax is presented in Backus-Naur Form (BNF) notation as shown in Figure 3.1. The corresponding syntax plots are shown in Figure 3.2 and Figure 3.3. The general syntax of a production rule is: name of the production (a non-terminal symbol), followed by the '=' symbol, followed by an expression, and terminated through the '.' symbol. In the figure, any text between pairs of '/*' and '*/' symbols are comments are ignored by the parser. The reader may already be familiar with the BNF notation from reviewing the syntax of ADA or other modern languages. ADA was defined by using the BNF notation. However, ADA does not possess an LL(1) grammar and is not LL(1) parsible. The BNF format used in this thesis differs somewhat from the USEWG document to allow compatibility with the parser program used by the author.

```
/* A language element is defined through a production rule. */
language_element = definition .

/* Keywords are shown in quotes and are capitalized for clarity. */
if_statement = 'IF' expression 'THEN' any_statement .

/* Definitions may contain optional parameters within square
   brackets. */
valve_type = 'THE' [ 'BIG' ] 'VALVE' .
/* valve_type will allow either 'THE BIG VALVE' or 'THE VALVE'
   to be parsed as syntactically correct. */

/* A logical 'AND' is indicated by having a symbol follow
   another.  For valve type in the last example, the 'VALVE'
   symbol must always follow the 'THE' symbol */

/* A logical 'or' is indicated using the '|' symbol. */
one_or_another = 'ONE' | 'OTHER' .
a_or_b_or_c = 'A' | 'B' | 'C' .

/* Operators of differing precedence are evaluated from highest
   to lowest.  Adjacent operators that are of the same precedence
   are evaluated from left to right. */
one_or_another_and_abc = one_or_another a_or_b_or_c .
/* The last definition permits one of the following:  'ONE A' or
   'ONE B' or 'ONE C' or 'OTHER A' or 'OTHER B' or 'OTHER C' */

/* Parentheses can override the left to right parsing. */
one_or_another_and_next = ( 'ONE' | 'OTHER' ) 'NEXT' .
/* The last definition permits only one of the following:
  'ONE NEXT' or 'OTHER NEXT' */
modified_one_or_another_and_next = 'ONE' | ( 'OTHER' 'NEXT' ) .
/* The last definition allows only one of the following:
   'ONE' or 'OTHER NEXT' */

/* Curly braces denote repeated elements. */
repeat_one_or_more_times = { ( 'A' | 'B' | 'C' ) } .
/* The last definition allows any of the following:   'A' or
   'AAA' or 'B' or 'BA' or 'CABBBA' but not '' (nothing). */

/* The '$' indicates where in the braces a repeat can exit */
repeat_zero_or_more_times = { $ ( 'A' | 'B' | 'C' ) } .
/* The last definition allows any of the repeat_one_or_more_times
   examples and also allows '' (nothing). */

loop_with_exit = { 'A' $ 'B' } .
/* The last definition allows one of the following:
  'A' or 'ABA' or 'ABABA' but not 'AB' or 'ABAB'. */
```

Figure 3.1 - Example BNF Notation

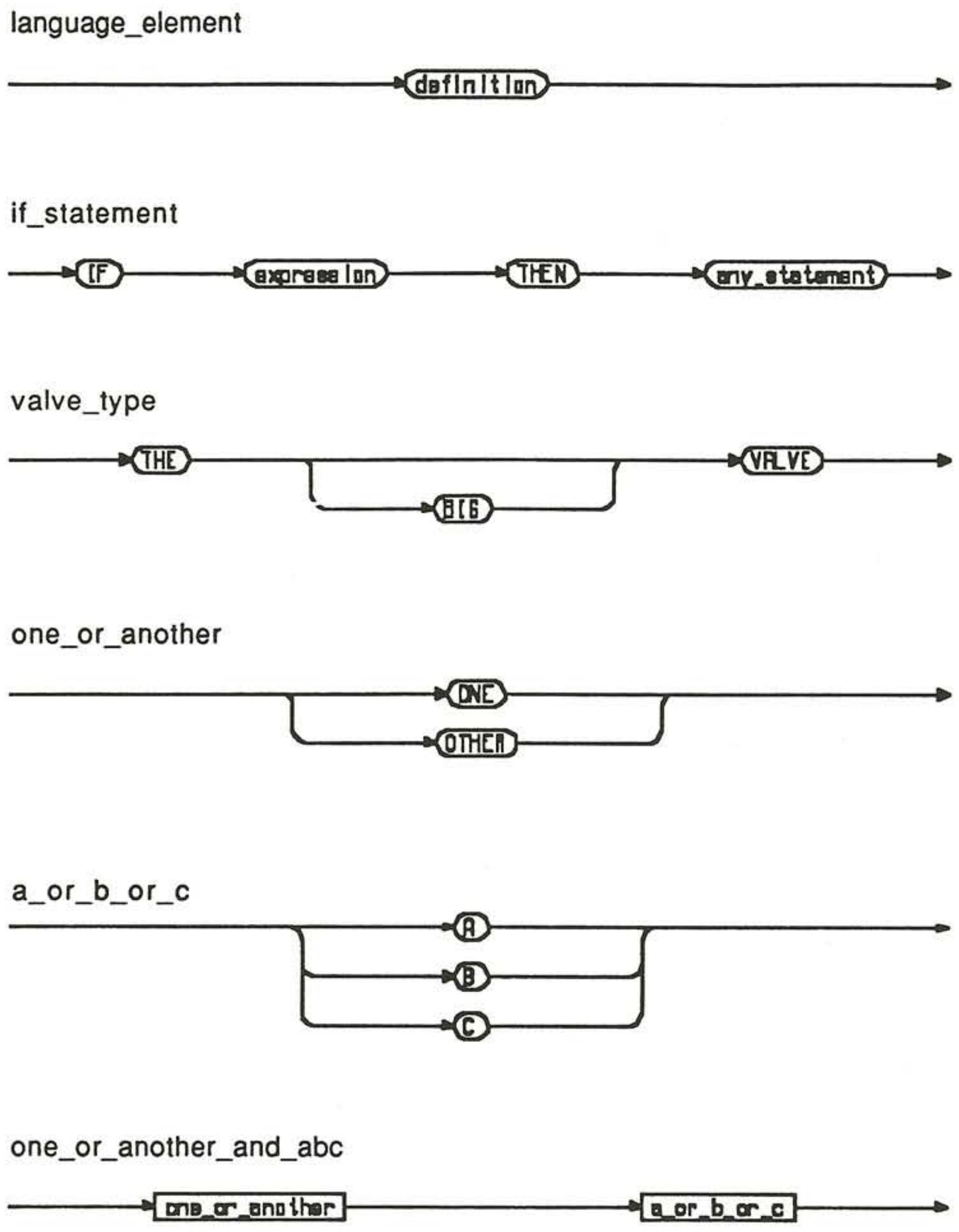
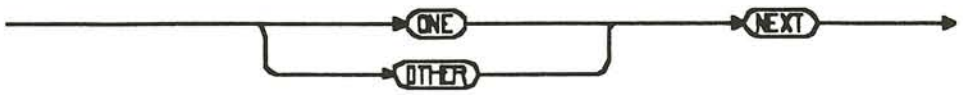
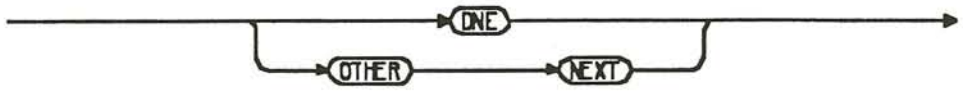


Figure 3.2 - Example Syntax Plots (1 of 2)

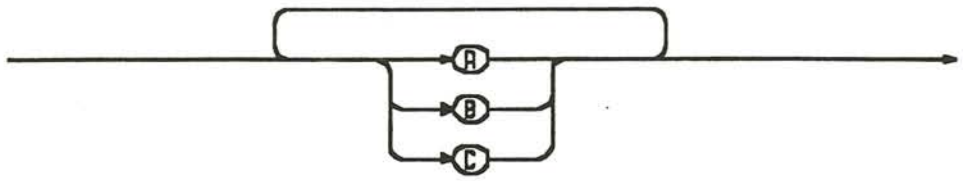
one_or_another_and_next



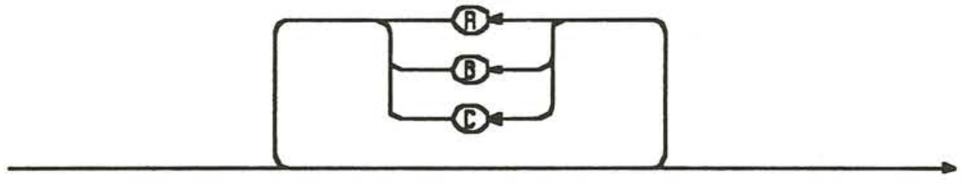
modified_one_or_another_and_next



repeat_one_or_more_times



repeat_zero_or_more_times



loop_with_exit

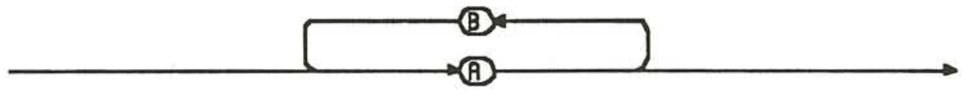


Figure 3.3 - Example Syntax Plots (2 of 2)

3.2. Character Set

The character set used for UIL is defined by the international standard ISO 646[24]. ASCII is the U.S. version of ISO 646. Except within strings, UIL treats characters as case insensitive.

3.3. Lexical Elements

UIL statements are composed with lexical elements, as constrained by the syntax, to form commands or procedures. The lexical elements are the words, numbers, strings, and other special symbols used by UIL.

3.3.1. Words

In UIL, a word is defined as shown in Figure 3.4. The USEWG UIL defines a word as letter { $ (letter | digit) }. The parser used by the author understands 'IDENT' as equivalent, and faster, than letter { $ (letter | digit) }. Therefore, 'IDENT' is used, and 'letter { $ (letter | digit) }' is commented out. The object_directory_entry has been added to the USEWG version to allow the parser to correctly recognize some typical UIL objects. Remember that these objects will be designed by NASA or NASA contractors for operational subsystems. The UIL executor and the parser must both know the objects and actions that the UIL procedure may use in order to verify the syntax of a UIL procedure.

```
word = 'IDENT' /* letter { $ ( letter | digit ) } */ |
object_directory_entry /* allows words to be OBJECT names */ .
```

Figure 3.4 - Definition of a UIL Word

3.3.2. Numbers

UIL supports the numeric lexical elements, such as integer or real, in the same format as ADA with minor exceptions. The USEWG version of UIL does not directly support complex data types, data structures or arrays. The lack of support for these data types may be a serious limitation for UIL programmers (see Chapter 5 for the author's data type extensions to UIL).

3.3.3. Dates and Times

A special data type is supported to represent dates and times in UIL. Dates and times are used extensively in the test and operation software that UIL is expected to be used for.

3.3.4. Strings

Unlike most languages, UIL defines two types of strings as shown in Figure 3.5. The text_string is defined in the same way as in most conventional languages. The pathname_string is a special string that represents only file pathnames. The pathname string represents a non-portable aspect of UIL. For example, the pathname_string '/usr/myfile' would be used in a Unix system; whereas, the pathname_string 'host1::dk0:[usr]myfile.' would be seen in a VMS system.

```
text_string = '"' { character } '"' .
pathname_string = ''' { character } ''' .

/* Examples
     text_string:       "Test one", "JSC 30497"
     pathname_string:   '/dev/mta3', 'usr:[test.progs]t13a.exe'   */
```

Figure 3.5 - Definitions of a String

The author believes that this is not the correct way to handle pathnames between different systems. This method builds into the language a feature that will result in the production of non-portable code. In Chapter 5, we shall propose a different method that will be portable.

3.3.5. Special Characters

UIL parallels standard languages by defining special characters for arithmetic operators or separators. Two examples of UIL special characters are '*' for multiply and '=' for assignment.

3.4. Basic Syntactic Components

This section describes the basic language structures of UIL that are built from the basic lexical elements defined in Section 3.3. The basic language structures are names, literals, lists, and expressions.

3.4.1. Names

UIL names are used to identify objects, object classes, attributes, and for specifying units of measurements. A name in UIL differs from most other languages in that a name may contain white space between the words within the name. All of the following are valid names: 'value', 'vacuum pump', 'pod bay door', and 'km per sec'. In the current USEWG version of UIL, the spaces are however not significant in determining whether two objects are the same, for example, 'pod bay door' is equivalent to 'podbay door'.

Each object accessible to UIL is identified by an object_identifier. A valid UIL object_identifier can be either an object name or a pathname. The pathname objects are created and maintained by the computer's operating system. An attribute_name is the name of an attribute within an object. In the example in Chapter 2, the port numbers in the object in Figure 2.4 were attributes. UIL provides access to the attribute by using the attribute_identifier syntax defined in Figure 3.6.

```
attribute_identifier = attribute_name 'OF' object_identifier .
```

Figure 3.6 - Definition of an Attribute Identifier

A UIL procedure that addresses the value of the open-input-port in valve 1 would use the phrase 'open-input-port of valve 1' (refer to Figure 2.4).

3.4.2. Literals

A literal is an explicit representation of the value of an object. UIL defines numeric, time, string, enumeration, and measurement literals. Numeric, time, and string literals provide the UIL programmer with constants. The enumeration literals represent the explicit values of the class of an object called 'enumeration' and are defined to be equivalent to the enumerated type in ADA. Measurement literals consist of numeric literals followed by a unit of measurement.

3.4.3. Lists

A list, as defined by the USEWG version of UIL, consists of zero or more objects separated by commas. The author believes that the list data type is very important, and that a more formal definition should be made. See how lists are used in Chapter 4 and the formal definition in Chapter 5.

3.4.4. Expressions

An expression is a formula that defines a computation. The rules for expressions in UIL are similar to those used in ADA or other conventional languages. Literals and objects are operands within the expressions that are acted on by operators. Operators of differing precedence are evaluated from highest to lowest. Adjacent operators that are of the same precedence are evaluated from left to right.

UIL provides function calls that can be used as operators for expressions like in other conventional languages. UIL also provides a method to refer to individual elements in an object by using indices. Figure 3.7 shows the definition of an indexed_object.

```
indexed_object = object_name '(' index_list ')' .
```

Figure 3.7 - Definition of an Indexed Object

Currently, UIL defines an index feature only for strings and lists. In the case of a string, UIL returns the characters that are specified in the index list. In the case of a list, UIL will return the items in the list that are referred to by the index list. The lack of support for complex data types may be a serious limitation for UIL programmers (see Chapter 5 for the author's data type extensions to UIL).

3.5. Statements

In UIL, statements are composed of lexical elements and syntactic constructs. There are six types of UIL statements defined by the USEWG version of UIL as follows: declaration, command, assignment, sequential control, conditional control, and iterative control. The version of UIL developed in this thesis has additional statements that control events and provide extensions for object and action creation. These new statements are described in Sections 3.8, 3.9, and 3.10.

3.5.1. Declarations

Declarations in UIL are used to create objects and classes. Figure 3.8 shows the syntax of the object declarations. The subclass declaration creates a new subclass of an existing object. The 'with' clause allows the instance attributes to be set. In the USEWG version of UIL, it is not possible to add new actions or disable existing actions using the subclass declaration. The object declaration is used to create a new object in computer memory. USEWG UIL only supports the creation of simple objects, such as integers or reals. The lack of support for UIL defined actions prevents programmers from using UIL for object based programming. The author believes that UIL must be able to create new objects, classes and actions to be useful on the Space Station project (see Section 3.10).

```
subclass_declaration = 'CLASS' object_name
                        'IS' object_name [ with_clause ] .

object_declaration = 'OBJECT' object_name
                      'IS' class_name [ with_clause ] .

with_clause = 'WITH' assignment { $ ',' assignment } .
assignment = ( name '=' expression ) .
```

Figure 3.8 - Syntax of Object Declarations

UIL also provides the following declarations: rename, parameter, and constant. The rename declaration provides an alternate name for an object and will primarily be used to assign an alias for an object with a long name. The parameter declaration describes the class and characteristics of the formal parameters (variables) in a procedure, for example, 'PARAMETER loop_number IS integer'. The constant declaration is used to create symbolic constants as in the following example: 'CONSTANT PI = 3.14159265358979'.

There are predefined actions for UIL objects. For example, the action 'new' is called when a new object is instantiated. The new action is responsible for the allocation and initialization of the new object. Once an object is instantiated, the object will exist until explicitly destroyed. When an object is no longer needed, the object is removed explicitly by issuing the destroy command.

Global objects in an object dictionary are available to users and event handlers, and they cannot be destroyed by a user. Objects created within a procedure are not visible to other procedures unless the object is passed to the other procedure.

3.5.2. Commands

UIL commands are used to control objects. Commands represent the way in which actions are sent to objects. The command is defined with more structure than simply 'action object' as can be seen in Figure 3.9. Remember that the actions and commanded objects are defined by the object directories that are active when a UIL procedure is executed. This means that the syntax does not have 'hard coded' values for specific object directories. For the purpose of the SSLO2 examples, the definition of the basic_command was modified by adding example_command. The definition of example_command contains all the SSLO2 specific action/object pairs.

```
command = basic_command [ qualifier_group ] .

basic_command =
     ( action ( commanded_object_list [ direction ] ) |
              ( direction commanded_object_list )    ) |
/**+ Added example object/action dictionaries */
     example_command .

commanded_object_list =  object_identifier_list [ 'OF'
                           object_identifier_list ] .
action = verb | ( 'START' verb ) | ( 'STOP' verb ) .
verb = word .

qualifier_group = { qualifing_clause $ ',' } .
qualifing_clause = ( qualifier expression ) | with_clause .
qualifier = 'AFTER' | 'AT' | 'BEFORE' | 'BY' | 'EVERY' |
            'FROM' | 'INTO' | 'TO' | 'UNTIL' | 'WHERE' .
```

Figure 3.9 - Syntax of Commands

For the SSLO2 example, example_command has been hard coded into the syntax to allow the syntax parser to be used for testing SSLO2 as shown in Figure 3.10. The actions and objects for most operational subsystems on the Space Station will be built and controlled by NASA. The run-time environment will have these names available for compiling and running UIL procedures. One can think of these in conventional terms as object directories and action dictionaries. The run-time environment will be able to check the syntax of UIL procedures only by knowing the object directories and action dictionaries that a procedure will require.

```
example_command = example_action example_object .
example_action =  action_dictionary_entry .
example_object =  object_directory_entry .

action_dictionary_entry =
/* events/systems */      'enable' | 'disable'   | 'stop'   |
                  'start' | 'revert' | 'enable_events_for'   |
/* operator */ 'write' | 'query'   |
/* systems */
        'chilldown_suction_line' | 'chilldown_transfer_line' |
        'chilldown_orbiter_mps'  | 'activate_topping'         |
        'open_main_fill_valve' .

object_directory_entry =
      'operator_object'    | 'fill_system_object' |
      'active_pump'        | 'backup_pump'        |
      'stop_key_object'    | 'revert_key_object' .
```

Figure 3.10 - Definition of SSLO2 Objects and Actions

3.5.3. Assignment Statement

The assignment statement modifies the value of a data object. UIL defines an assignment statement in a fashion similar to that of the BASIC language. A 'let' keyword precedes the assignment to improve the readability and clarity.

3.5.4. Sequential Control Statement

Sequential control statements are used to control the execution of UIL procedures. UIL defines the following sequential control statements: null, step, goto, wait, and return. The goto and return statements provide functions similar to those that are found in most conventional languages. The null statement performs no operation. It can be used to provide as a placeholder for a statement when no operation is desired. The step statement allows a label to be assigned to a sequence of UIL statements. This label is used as the destination for a goto statement. The wait statement suspends a UIL procedures execution until a logical expression becomes true.

3.5.5. Conditional Control Statement

Conditional control statements are used to test logical expressions and change the flow of execution. The conditional control statements in UIL consist of 'if', 'verify', and 'case'. The 'if' and 'case' statements parallel those found in other languages. The verify statement is unique to UIL. It is modeled after an if statement with the addition of the 'WITHIN' timing_simple_expression clause as shown in Figure 3.11. If the logical_expression does not become true within the time limit, then the otherwise clause is taken. The raise statement described in Section 5.1.6 provides a convenient method of handling exceptions.

```
verify_statement =
              'VERIFY' logical_expression
                  [ 'WITHIN' timing_simple_expression ]
              [ 'THEN'
                 sequence_of_statements ]
              [ 'OTHERWISE'
                 sequence_of_statements ]
              'END' 'VERIFY' .
```

Figure 3.11 - Definition of a Verify Statement

The verify statement has been added primarily to make the porting of GOAL procedures into their UIL equivalents convenient. The verify statement sometimes reads better than the if statement for control applications and has the added timing feature. In GOAL, the verify statement acts on external function descriptors, whereas the if statement acts on logical expressions.

3.5.6. Iterative Control Statements

Iterative control statements are used to test logical expressions and repeat the execution of other UIL statements. UIL defines several iterative control statements, including a repeat, while, for, and exit statement. They perform similar functions to those found in other conventional programming languages.

3.6. Execution Environments

This section in the USEWG document discusses several commands that are used to invoke UIL environments. In this thesis, we do not address these commands as they are not part of the language syntax.

3.7. Procedures

UIL procedures are defined in a similar manner to procedures in conventional languages. A procedure may be compiled and saved. A procedure may carry arguments that allow information to be passed into the procedure for execution. The USEWG document also discusses several commands that are used to invoke UIL procedures that are not addressed by this thesis.

3.8. Events and Event Handlers

An event is used to notify a procedure of a change in the condition of hardware. The event is the hardware signal. When this signal occurs, an event handler programmed to respond to the event will be executed by the computer. The USEWG definition is shown in Figure 3.12. The USEWG version of UIL does not allow event handlers to be enabled or disabled by a UIL program. The GOAL version of SSLO2 makes extensive use of event handlers; therefore, the version of UIL used in this thesis supports the use of event handlers from within UIL procedures.

```
event_handler = 'EVENT' 'HANDLER' 'FOR' event_name
                'IS' sequence_of_statements
                'END' [ event_name ] .
```

Figure 3.12 - Definition of an Event Handler

3.9. UIL Extensions for SSLO2

Extensions have been added to the USEWG version of UIL in a way to impact the syntax of UIL as little as possible. The definition of UIL was extended by adding declarations, event_handlers, and declaration_extentions as seen in Figure 3.13.

```
UIL = { /* USEWG UIL */ procedure | declaration |
        /* new */      event_handler | declaration_extentions } .
```

Figure 3.13 - Definition of UIL

The declarations allow UIL to be used to define objects that are not in a procedure, as in the case of global objects. The event_handler extension allows UIL to handle external events, such as interrupts from hardware, in a similar way as the GOAL version of SSLO2 handled interrupts. The declaration_extentions allow complex objects to be created by UIL. The declaration_extentions are defined in detail in the next section.

3.10. Object Description Language Extensions for SSLO2

A major extension to the USEWG UIL proposed in this thesis is the support for creation of complex objects. UIL currently only allows the manipulation of complex objects and the creation of only very simple objects such as integers and real numbers. In a way, it is incorrect to call the USEWG UIL a full-fledged object oriented language, as it is really an object manipulation language only. The USEWG version of UIL can only be used to send actions to objects and interpret the results.

In Figure 3.14 and Figure 3.15, the extensions to the UIL syntax are shown that will make UIL a full-fledged object oriented language. There are several functions needed for this purpose, namely the possibility to define actions, classes, and objects, and a statement to 'send' an action to an object.

```
declaration_extentions =
     ( 'DEFINE' ( big_object | big_class)  ) | send_action .

big_class =     'CLASS' name_list [ 'IS' name_list ]
      { $ declaration |
         ( 'ACTION'    ( name | action_dictionary_entry )
           'IS' ( ( name | action_dictionary_entry ) | 'NULL' ) ) }
      { $ uil_action }
   'END' 'DEFINE' 'CLASS' [ name_list ] .

big_object =
    'OBJECT' { name_list } 'IS' name_list
      { $ declaration |
         ( 'ACTION'    ( name | action_dictionary_entry )
           'IS' ( ( name | action_dictionary_entry ) | 'NULL' ) ) }
      { $ uil_action }
   'END' 'DEFINE' 'OBJECT' [ name_list ] .

uil_action =
   'DEFINE' 'ACTION' name [ '(' parameter_name_list ')' ] 'IS'
      { $ sequence_of_statements }
   'END' 'DEFINE' 'ACTION' [ name ] .
```

Figure 3.14 - Definition of Complex Objects and Classes

The big_class definition begins with the name of the new class to be created, and with the (optional) 'IS' clause that allows inheritance of actions and instance variables from another class. The big_class definition also contains the declarations of object attributes, logical translations from external action names to the internal uil_action procedures, and the actual uil_action procedure definitions. The action name translation will also allow properly written ADA procedures to be called instead of an internal uil_action procedures. The big_object definition allows the creation of a new object by class name. The declaration and uil_action portion of big_object allows the ability to add or override any class declarations and uil_actions. The uil_action definition defines a UIL procedure that will be linked to the object as an action. When an action is 'sent' to an object, this is actually the UIL procedure that will be called. The uil_action is defined in the same way as any UIL procedure would be.

While global UIL object directories and action dictionaries will be available before run-time, the new actions created by big_class and big_object will not. Therefore, a new statement called send_action is defined in Figure 3.15 to provide a formal syntax for sending actions to objects. Both forms of the send_action statement shown below are functionally equivalent - choose the one that fits better for any particular action and object pair.

```
send_action = ( 'SEND' an_action 'TO' an_object) |
              ( 'ASK'  an_object 'TO' an_action) .

an_action = ( action_dictionary_entry | action ) [ with_clause ] .
an_object = 'SELF' | object_name .

/* Usage examples:   SEND pump_status TO operator
                     ASK  valve_123   TO open       */
```

Figure 3.15 - Definition of an Formal Action/Object

These forms allow any action and object to be used without conflicts or parsing problems, thereby allowing a UIL procedure to be verified as syntactically correct. It is expected that developers will only use send_action forms for non-NASA controlled objects classes (For example, within programs that use complex objects that are not exported outside the application). In the case of the SSLO2 example, all objects were created by the statements defined in this section.

4. EXAMPLES OF UIL USAGE

This chapter discusses the SSLO2 example used by the author to verify that a version of UIL could be used to replace GOAL. Before discussing SSLO2, we shall discuss some of the issues that constrained the GOAL programmer, and we shall review GKH1F, a much simpler GOAL open valve routine. The parser program is used to check whether the ported GOAL code is syntactically correct UIL code. The reader is referred to Appendix B for the complete listing of GKH1F, to Appendix C for a listing of the high level control of SSLO2, and to Appendix D for a listing of the SSLO2 action and object definitions.

4.1. GOAL code example

What type of problems is the GOAL programmer faced with? The answer to this question should provide some insight into how UIL should be designed. The GOAL environment is quite primitive by today's standards; however, remember that the Shuttle program found its roots more than ten years ago. In the same respect, decisions made for Space Station Freedom now will influence the U.S. Space program for the next 10-20 years.

4.1.1. The GOAL Environment

The environment in which GOAL programs execute is quite different from what you might imagine. The GOAL compiler was written to support the special requirements of the controlling hardware. The operating hardware is quite simple and very restrictive.

Like many other special purpose languages, GOAL code is compiled into an intermediate level code. The intermediate code is, in turn, interpreted by the GOAL executor when the operator invokes the 'task'. These tasks are executed on a special workstation built for NASA, hereafter simply called a console. A console can run only up to four tasks simultaneously, each of which can have up to four levels (subroutine nesting levels). Each task has access to not more than six hardware supported timers that can be used to time e.g. valve open and close times, and program time-outs. Each task can also access up to six disk files, each file is 512 words in length. It is likely that a single task would control a complete subsystem.

The task main loop code does not perform many time critical operations. Errors and exceptions are typically handled by interrupt handlers (also written in GOAL). For example, it is not uncommon to see delays of one or two seconds in the main command loop code. For a task, interrupts are handled by the lowest active level and are queued if received by that active level and enabled only at a higher level than the current execution level.

The console itself consists of a single CPU, three color displays, three keyboards, two moving head disk drives (ten MBytes each), a printer/plotter, fifteen PFPK keys (programmable keypad buttons), and memory to store up to seven application pages of display information (one viewable, six in background memory) for each display in the console.

4.1.2. Operational GOAL Procedures from KSC.

KSC provided listings of six GOAL procedures as shown in Table 2. All of the procedures were reviewed, and the SSLO2 and GKH1F procedures were studied in detail.

The procedures provided a wealth of information and insight into the unique problems of controlling hardware systems.

Table 4.1 - GOAL Programs from KSC

Procedure	Statements	Function
SSLO2	2998	fills the LOX portion of the external tank
GAH51	5210	monitors and updates status of the LH2 system
GKL4W	114	opens the A86460 LOX replenish valve
GKL4X	126	closes the A86460 LOX replenish valve
GKH1F	68	opens the A100677 LH2 main fill valve
GKH1G	62	closes the A100677 LH2 main fill valve

Several interesting general comments can be made relating to GSE (Ground Support Equipment) and Orbiter hardware design as observed in the GOAL procedures reviewed. The software reviewed represents less than one task in a console. Some observations that can be made from the code reviewed are:

(1) The more important devices (such as valves or pumps) have redundant units that can be switched in to replace the primary device in the event of failure. Additionally, many important sensors (pressure, temperature, etc) have redundant sensors that can be used to verify the results of the primary.

(2) The operator typically verifies every major step completed by the procedure, including the selection of a primary or redundant main device (such as a pump). Most exceptional conditions that arise cause the procedure to suspend execution to wait for a 'continue' or 'stop' command from operator. (Interrupt handlers are not disabled while the operator responds.)

(3) Most sensors have bypass flags that can be set to inhibit their use. This flag is in a shared memory and is set by the operator in response to a hardware failure. This flag must be explicitly checked by the GOAL procedure prior to the use of the sensor.

(4) In several cases, GOAL procedures used voting on sensor information to determine if an operation is 'safe' (e.g., two out of three sensors show a pressure nominal). The GOAL code that is used to vote on sensor information is hard coded into the procedure. This code is made even more complex by the bypass flags.

(5) Many hardware related constants are imbedded directly into the code (e.g., valve open delay times, maximum temperature, and conversion constants).

(6) Some pumps controlled by GOAL code have tachometers. When a new motor speed is desired, the program outputs the speed to the motor and delays execution for a predefined time. At the end of this time, the program checks the pump speed to determine whether the motor has reached the desired speed.

(7) If the pump has not reached the desired speed, then the GOAL procedure tries to 'control' the motor using software. Finally, if that does not appear to work, the operator is given the choice to continue procedure execution, or to override any apparent problem indicators.

(8) Many instrument measurements read from hardware are not in the correct units (e.g., the LOX temperature used in the code is the returned temperature value plus -293 degrees F). It is the programmer's responsibility to adjust the readings to consistent units.

(9) The GSE is different between launch pad A and launch pad B. These differences are hard coded into the GOAL procedures. The differences often add additional complexity to the GOAL procedures.

(10) It is also possible that Orbiters (or payloads) have different GOAL code. The code reviewed did not have an example of this; however, the code is changed almost one out of every two Shuttle flights. One revision comment was found to refer to the deletion of a check for the OV99 tail number (Orbiter Vehicle number 99). OV99 was the Challenger, and the comment was removed two years before its destruction.

(11) Most sensors can be programmed to generate hardware interrupts to the GOAL program. In fact, interrupts appear to be the preferred method of getting information. In cases where analog measurements are possible, interrupts can be generated on reaching a low and/or a high limit.

4.1.3. GOAL program design

The console used for the Space Shuttle provided a limited environment for GOAL program development. The effects of the console limitations (and the technology available in the implementation phases of the Space Shuttle program) can be witnessed in the SSLO2 code. Due to the limit of four subprogram levels (and execution speed concerns), GOAL programs rarely used subroutine calls. This had the effect of creating long, hard to understand programs. One of the few exceptions to this rule was valve open and close routines.

Macros are used quite often to substitute for similar code segments. For example, SSLO2 makes use of a macro to ramp the main pump about eight times. This generates many duplicate sections of the same code. In the case of the main pump, one macro expansion was used for each pump and pad: 126 on pad A, 126 on pad B, A127 on pad A and A127 on pad B. The same macro was also used several times in the 'revert' recovery section. Other interesting observations:

(1) SSLO2 contains 2998 GOAL statements. Each statement used from one to eight text lines in the listing (typically four lines).

(2) About 850 statements (28%) of SSLO2 end with GO TO STEP. There were about 640 statement labels.

(3) 26% of the SSLO2 code is used to set up or handle interrupts.

(4) 24% of the SSLO2 code handles recovery from an operator 'stop' or 'revert' command.

(5) The SSLO2 code references 450 different external function designators. These external function designators refer to the hardware valves, pumps, switches, printers, plotters, lights, and so on.

(6) Often there are many external function descriptors specified in a single GOAL statement. (For example, CHANGE <FD1> <FD2> <FD3> <FD4> <FD5> ... <FD9> SAMPLE RATE TO 0 TIMES PER SECOND). The SSLO2 program would be 30-40% bigger if GOAL did not provide a list capability.

(7) Shared memory, external to the console, may be used to hold current state information of the execution of GOAL procedures. For example, <NLOK3000X $LOX ANALOG COUNT$> is apparently used to hold the 'state' of the SSLO2 (and of its parent program) in memory outside the console CPU.

(8) SSLO2 code references 109 externally defined subprogram calls, mostly valve open and close procedures.

4.1.4. GOAL Compiler Operation

The GOAL compiler helps make the code more readable by expanding cryptic function descriptors by adding comments. For example, the cryptic external function designator <GLOQ1219A> is translated to <GLOQ1219A $A37334 PMP A127 CLTCH H2O LOWRATE$> on the output listing. The GOAL compiler, however, fails to expand cryptic program descriptors (For example, GKL4W is not shown as GKL4W $Open A86460 LOX replenish valve$). The GOAL compiler also fails to clearly show nesting levels (e.g., within if/then/else).

The GOAL compiler took thirty minutes elapsed time or seven minutes CPU time to compile 2998 lines of code (machine type unknown). In the event of emergency changes to a procedure, this amount of time may be to long and may prohibit its use for real-time changes in the configuration of equipment. By comparison, modern C compilers on better personal computers can compile code at up to 100K lines/minute.

4.1.5. The GOAL GKH1F procedure

The SSLO2 procedure is too long to reproduce in its entirety in this thesis; therefore, the much shorter GKH1F procedure will be used to describe how GOAL code is interpreted. The complete GOAL listing of GKH1F is contained in Appendix B. Section 4.3.2 describes the UIL version of the GKH1F. The procedure GKH1F contains 68 GOAL statements. There are about 120 lines of user documentation, mostly contained in the header. Of tne 68 GOAL statements, two statements are used to turn the valve on, twenty statements are used to time the duration of the valve opening, nine statements are used to display status information, leaving the remaining 38 statements to check or override safety indicators. Section 4.3.2 describes the UIL version of the GKH1F.

As defined in the GOAL syntax, all items enclosed by '<' and '>' are external function designators. For example, <GLHX4112E> is a valve closed indicator that becomes true when the valve hardware is in a closed state. The GOAL compiler adds comments (enclosed by $....$) for each external function designator in the source code. The designator appears in the output listing as '<GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>'. In GOAL, items enclosed between '(' and ')' refer to variables used by the procedure.

Several code fragments have been extracted from the listing of GKH1F for the discussion below. The program begins with a long header comment that describes who wrote it, what it will be used for, its revision history and other information as shown in Figure 4.1.

```
$         The program is designed to command the A100677 main
          fill valve to the open position and set the appropriate
          exception monitor and GOAL notification limits.

          The following types of function designators are defined
          in the data base and are used in this program-

          <GLHK----E> Valve discrete commands
          <GLHX----E> Valve discrete position indicators
          <NLH------> Bypass function designators

          The respective bypasses are turned on whenever it is
          desired to inhibit the issuance of the command(s)
          to the component or if component response indicator(s)
          and/or signal(s) is determined erroneous. No bypasses
          are turned on within this program.$
```

Figure 4.1 - Some GHK1F Header Comments

The program then changes the interrupt handlers as needed to perform the function to open the valve as seen in Figure 4.2. None of the interrupt commands take effect until the 'ACTIVATE INTERRUPT PROCESSING ON THIS LEVEL' command is executed. The 'SPECIFY INTERRUPT' statement defines the action the program will take when the operator pushes the key '6' on the console; in this case, the command aborts the GKH1F procedure and exits. The 'CHANGE SAMPLE RATE' statement modifies the rate at which the signals of importance to GKH1F are monitored.

The next two statements inhibit exceptions from being raised when the indicators change due to normal operation of the program. Finally, the GHK1F procedure informs the system of the condition that the indicators should have after the program terminated. The bypass flags are software flags that are stored in system common memory. These flags are changed by the operator to allow operation of the software when the indicators are not functioning properly.

```
   SPECIFY INTERRUPT <PFPK6 $PSP KEY 6 DEFAULT$>
        AND ON OCCURRENCE GO TO STEP  40;

-several statements omitted from the example at this point-

    CHANGE <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$>
         <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$>
         <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
         <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>
         <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
         SAMPLE RATE TO 100 TIMES PER SECOND;

    INHIBIT EXCEPTION MONITORING FOR
         <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
         <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
         <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>;

    INHIBIT FEP INTERRUPT CHECK FOR
         <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
         <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
         <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>;

$   The previous statements inhibit those interrupts which
    would result from this valve changing state.  All other
    interrupts are activated.  $

$   The following statements check each valve position indicator
    bypass.  If the bypass is on, no action is taken.  If the
    bypass is off, GOAL and system exception conditions are
    changed to reflect the state each indicator is expected to
    show upon program completion.          $

    VERIFY <NLHX4112E $A100677 MAIN FILL VALVE CL IND BYP$> IS OFF
         ELSE GO TO STEP   2;

    CHANGE <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
         GOAL EXCEPTION CONDITION TO ON;

    CHANGE <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
         SYSTEM EXCEPTION CONDITION TO ON;

-several statements omitted from the example at this point-

STEP 100  ACTIVATE INTERRUPT PROCESSING ON THIS LEVEL;
```

Figure 4.2 - Enable Exceptions

Now that the interrupt processing has been activated for the operation of the GKH1F program, the valve can be opened as shown in Figure 4.3. The commands to open the valve are sent using the first two statements, and the command is verified by the third statement. If the 'VERIFY' statement does not indicate that the commands were sent, an exception is raised to the operator.

```
$   The valve is commanded to its open position and the program
    verifies that prerequisite control logic allowed the commands
    to be issued.          $

STEP   5
    TURN ON <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$>;

    TURN OFF <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$>;

    VERIFY <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$> IS ON
      AND <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$> IS OFF
         ELSE GO TO STEP   15;
```

Figure 4.3 - Open Valve Command

Execution of GKH1F is far from complete at this point. Figure 4.4 shows that the procedure must monitor the opening of the valve and verify that it is open in the expected time. The program reads in the current time and prepares to measure the time of the initial component motion, that is, the time it takes the valve closed sensor to become false.

The timing loop is hand coded for each timed event. The programmer must be careful to specially consider the case where the timer starts before midnight GMT and ends after midnight GMT. In that case, 86400 seconds (the number of seconds in a day) must be added to the measured timer value. The operator and a printer log is informed of any exceptional condition. Only the initial component motion timing loop is shown in Figure 4.4. The GKH1F procedure also verifies that the valve finally depresses another sensor which detects that the valve has fully opened.

```
$   The following timing loop allows 6 seconds in which to
    establish initial component motion and up to 20 seconds
    for the valve to home.     $

    READ <GMT $GREENWICH MEAN TIME$> AND SAVE AS (GMT1);

STEP   6
    READ <GMT $GREENWICH MEAN TIME$> AND SAVE AS (GMT2);

    LET (VLVTM) = (GMT2) - (GMT1);

    IF (VLVTM) IS LESS THAN 0.0 SEC,
         LET (VLVTM) = (VLVTM) + 86400 SEC;

    IF (VLVTM) IS GREATER THAN OR EQUAL TO 6 SEC,
         THEN GO TO STEP    9;

    VERIFY <NLHX4112E $A100677 MAIN FILL VALVE CL IND BYP$> IS OFF
         ELSE GO TO STEP    7;

    VERIFY <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$> IS OFF
         ELSE GO TO STEP    6;

    GO TO STEP 10;

STEP   7
    VERIFY <NLHX4123E $A100677 MAIN FILL VLV RED IND BYP$> IS OFF
         ELSE GO TO STEP    8;

    VERIFY <GLHX4123E $A100677 MAIN FILL VALVE RED IND$> IS OFF
         ELSE GO TO STEP    6;

    GO TO STEP 10;

STEP    8  RECORD <GMT $GREENWICH MEAN TIME$>
         FORMAT (NO UNITS,NO FD NAME,NO FD DESCRIPTOR)
         , TEXT ( GKH1F- VLV A100677 CLOSE IND GLHX4112E AND RDCD)
         NEXT TEXT (       IND GLHX4123E ARE BYP)
         NEXT TEXT (       INITIAL MOTION CANNOT BE DETERMINED)
         TO <PAGE-A $DISPLAY APPLICATION PAGE B$> YELLOW
         TO <CNSL-PP $CONSOLE PRINTER PLOTTER$>
         <SPA-PRNTR $SPA PRINTER$>;

    GO TO STEP 10;
```

Figure 4.4 - A Timing Loop

Once the valve has been opened, or once the operator has overridden any exceptional condition, the program terminates successfully. Before termination, the procedure must activate changes in exception handling procedures on any indicators that have new values, and deactivate any GKH1F procedure sampling of indicators (see Figure 4.5).

```
$   Interrupts are activated for those valve indicators which are
    not bypassed.       $

    VERIFY <NLHX4112E $A100677 MAIN FILL VALVE CL IND BYP$> IS OFF
         ELSE GO TO STEP   17;

    ACTIVATE EXCEPTION MONITORING
         FOR <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>;

    ACTIVATE FEP INTERRUPT CHECK
         FOR <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>;

-several statements omitted from the example at this point-

STEP  19  CHANGE <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$>
         <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$>
         <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
         <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>
         <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
         SAMPLE RATE TO 0 TIMES PER SECOND;

-several statements omitted from the example at this point-

    TERMINATE;

    END PROGRAM;
```

Figure 4.5 - Disable Events

4.2. Porting SSLO2 to UIL

In the USEWG UIL specification, one thing was apparent: The USEWG UIL specification does not allow a UIL program to create new complex classes or objects (see Section 3.5.1 or 2.2.4 of the USEWG document). There is no syntax provided to describe and/or build complex classes or objects. All but the simplest classes are assumed to be built by ADA programmers and provided, via the operating system, to the UIL executor. Similarly, there is no discussions about class inheritance or the issues on what to do with (or how to control) the object databases. The assumption that UIL will not be used to create complex objects or classes simplifies the syntax and implementation of UIL; however, it prevents UIL from being used to develop any complex objects or even be used for object based programming. It can only manipulate existing objects and classes.

While this may be the desired UIL language, it is inadequate to port the GOAL version of SSLO2 into UIL. UIL, as currently described, cannot be used to completely describe the functionality contained in SSLO2. SSLO2 is comprised of low level instructions (for controlling registers on hardware), and of higher level instructions (used for the control of the loading of the external LOX tank). All of the code in SSLO2, whether low or high level, was written in GOAL.

The USEWG UIL specification only covers writing high level control procedures. Low level control procedures and class definitions must occur somewhere else (perhaps in ADA code or, more likely, using object builder tools written in ADA).

Three areas of object programming can be identified in the SSLO2 example:

(1) Writing high level software to control the overall process. This consists of conventional statements that act on objects, e.g. 'if' and 'for' statements.

(2) Building the object description of hardware to be controlled, for example, instantiating the objects that make up the actual hardware.

(3) Developing the classes of objects (pumps, motors, etc...) that will be used to instantiate the object description of the hardware. This work includes defining both the object attributes and the object actions.

The USEWG UIL specification is correct and useful for (1), can be used for (2), and is not useful for (3). Considering that UIL will be used for the next 15-20 years for Space Station development, UIL should provide a complete object programming environment. UIL will be used for programming applications like SSLO2 and many others from a wide range of areas that Space Station developers, testers, and users will encounter. UIL procedures should be able to describe, create, and destroy complex classes or objects with associated actions and event handlers as needed. The question whether or not a particular UIL user is granted permission to create such objects or object classes in UIL should be resolved by user privileges, and not by the inadequacy of UIL.

In the next few sections, the reader will be shown how the extensions that have been added to UIL will allow a complete object based solution for problems such as SSLO2.

4.3. SSLO2 in UIL

A part of SSLO2 was ported from GOAL to UIL and will be discussed in the following three sections. In Section 4.3.1, the UIL version of the top level control functions from SSLO2 are described. Section 4.3.2 describes the UIL version of the GKH1F open valve procedure written using object based techniques. Section 4.3.3 describes the object structure used to represent the hardware system that SSLO2 controls.

This section reviews the hardware GSE that the SSLO2 procedure controls. The GOAL listing of SSLO2 was over 200 pages long, and therefore, it is not included as an Appendix to this document. The UIL version of SSLO2 is described in the following three sections that parallel the three areas of object programming identified in the previous section.

Figure 4.6 shows a simplified block diagram of the GSE that SSLO2 was written to control. The GSE is composed of a primary and an alternate LOX pumping subsystem. These pumps are used to transfer over 1,359,000 pounds (143,000 gallons) of LOX from a storage tank to the Shuttle external tank before a launch. Each pump has an associated motor, a clutch and a clutch cooling system housed in an equipment room. In the figure, the items from the alternate pumping subsystem are identified with lower case 'a'. Various sensors are also shown in the drawing: 'T' sensors measure temperature, 'R' sensors measure rotational velocity, and 'L' sensors measure the level of a tank.

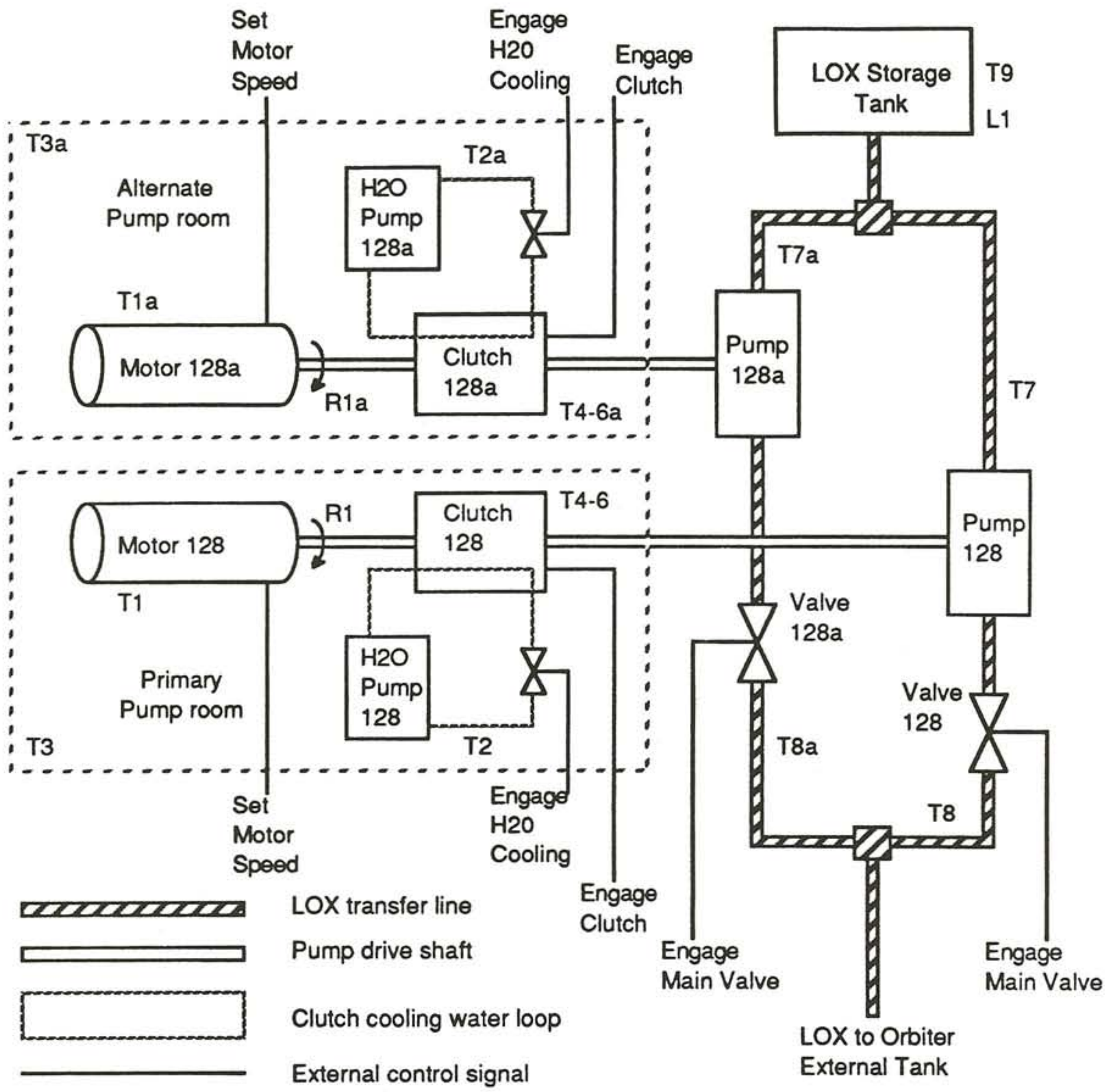


Figure 4.6 - Orbiter External Oxygen Tank GSE

The external control lines shown in the figure are the control signals that the SSLO2 procedure sets to actuate the proper hardware. The sensors shown in the figure are inputs to the procedure to verify that operations are executing correctly. There are many other sensors that are not shown which monitor whether a valve is open or closed, or whether a signal is set to the correct value. The hardware controlled is massive indeed. In the case of the main valve, it can take from 15 to 30 seconds to completely open. The size and the mission critical nature of the hardware help explain why it takes over 3000 lines of GOAL code to fill a single tank. UIL will provide a simpler and more consistent approach to controlling hardware on the Space Station.

4.3.1. Top Level Control Procedure

In this section, the UIL code that will perform the high level functions of SSLO2 will be reviewed. The most significant difference from the GOAL version of SSLO2 is that, in the UIL version, code responsible for the high level functionality can be separated from the lower level functions, whereas, in the GOAL version, the high and low level functions cannot be separated. The UIL version relies on the objects to perform low level functions leaving the high level code uncluttered. In the UIL version, the high level code accounts for about 40 UIL statements. How many lines of GOAL code account for the high level functions cannot be determined. The presented UIL version of SSLO2 does not translate all the functionality of the original 3000 line GOAL program; however, it reflects a representative core of the original program (perhaps 30-40%). The complete listing of the high level UIL code that was written is given in Appendix C.

The GOAL procedure, SSLO2, is the Space Shuttle LOX Auto Fill Sequencer. The procedure performs the operations necessary to fill the LOX portion of the Shuttle external tank. It remains in execution until the 100% liquid level sensors flash wet, that is, show that the tank is 100% full. The first operations performed by SSLO2 are to set up the event handlers and perform instrumentation status checks. Next, SSLO2 performs a chilldown of the following items: LOX pump suction line, pump transfer line, and orbiter MPS (Main Propulsion System). The chilldown step is performed to slowly and carefully reduce the temperature of the hardware to the temperature of LOX, about -186 degrees. Once chilldown has been completed, SSLO2 performs the following steps: a slow fill of external LOX tank to 2%, fast fill of external LOX tank to 98%, and a slow topping fill of the external LOX tank to 100%. SSLO2 then transitions to a replenish program to keep the external tank topped off to between 100% and 100.15%.

For all the example code segments shown in the following sections, the following rules apply: UIL language elements are capitalized, names use '_' between words, class names are terminated by '_class', object names are terminated by '_object', strings are marked with double quotes, and enumeration constants are marked with single quotes. Although the notation differs somewhat from what the actual UIL code would look like, it is intended to make the examples easier to read.

Figure 4.7 shows the procedure definition for SSLO2. It has a similar format to most conventional procedural languages. The 'CONSTANT' declaration defines a constant called long_name to be equal to the value of the string "Program unit SSLO2". The 'PARAMETER' declaration defines a variable called active_pump that will be used to hold the pointer to an object of the class hwsubsystem_class. The declaration of sslo2_status defines an ENUMERATED data type that can only be assigned the values 'run', 'stop', or 'revert'.

```
PROCEDURE sslo2_procedure IS
   CONSTANT  long_name = "Program unit SSLO2"
   PARAMETER active_pump    IS hwsubsystem_class
   PARAMETER backup_pump    IS hwsubsystem_class
   PARAMETER temp           IS basic_object_class
   PARAMETER sslo2_status   IS ENUMERATED
                            WITH VALUES = {'run', 'stop', 'revert'}
```

Figure 4.7 - SSLO2 Procedure Definition

Similarly to the GOAL version of SSLO2, the next few statements in the UIL version of SSLO2 perform the setup of event handlers. The event handlers shown in the next figure are used to allow operator interruption of the fill process. If the operator wishes to stop the fill operation, he can press a 'STOP' key, and thereby put the fill operation on hold. Once the stop key has been pressed, the operator can either 'REVERT' or cancel the fill operation. The revert option would reverse the stop, that is, continue the fill operation from where it had been interrupted.

The first statement, 'disable stop_key_object', in Figure 4.8 is an object/action pair. The object stop_key_object represents a push button that the operator can push. It may be a large red button on the operators console or just a button in a window on the terminal. SSLO2 does not need to know what kind of a button it is because the stop_key_object itself contains the detailed button control code. What is important is that we can just send the button the 'disable' action. This action disables any prior event handler from being activated should the operator push the button. SSLO2 disables both buttons in preparation for changing event handlers.

```
disable stop_key_object
disable revert_key_object

EVENT HANDLER FOR stop_key_object IS
      enable    revert_key_object
      disable   stop_key_object
      stop      fill_system_object
      LET       sslo2_status = 'stop'
      query     operator_object WITH options = {'OK', 'quit'}
   END stop_key_object

-several statements omitted from the example at this point-

enable stop_key_object

enable_events_for fill_system_object
```

Figure 4.8 - Setup of Event Handling

The next statements install the event handlers for SSLO2. In this case, the 'EVENT HANDLER FOR' statement installs the procedure shown between 'IS' and 'END'. When the stop key is pushed, this procedure will be executed. The procedure performs the following operations: enable the revert key, disable the stop key (we are already stopped), change the procedure status, write a message to the operator that the stop has happened, and wait for the operator to acknowledge. It is assumed that the stop fill_system_object would have informed the operator of the stop condition by the time the query operator asked for an 'OK'.

This event handler is obviously incomplete. Some of the other operations that have to be performed in a stop event would be: determine the current state, correctly shut down the fill operation, and put the equipment into a hold for a cancel or revert operation. These operations could be in the objects or in the top level procedure depending on the the overall design. For comparison, almost 25% of the GOAL version of SSLO2 was associated with handling just the stop and revert operations.

Continuing into SSLO2, Figure 4.9 shows the interaction with the operator to select whether the primary or the alternate pumping system will be selected. In the first statement, the write action sends an output string to the operator indicating that execution of the SSLO2 procedure has started. The next statement queries the operator to select the pump subsystem to use for this fill operation and waits for a response. After the operator responds, the procedure selects the correct pump from the fill_system_object. This is the first example of the use of an object attribute.

```
write operator_object WITH output =
   "SSLO2 procedure execution has begun"

query operator_object WITH output  = "Select main p128 pump or
                                       alternate a128 pump",
                           options = {'main', 'alternate', 'quit'}

IF response OF operator_object = 'quit' THEN RETURN END IF

IF response OF operator_object = 'main'
    THEN
       LET active_pump = main_pump      OF fill_system_object
       LET backup_pump = alternate_pump OF fill_system_object
    OTHERWISE
       LET active_pump = alternate_pump OF fill_system_object
       LET backup_pump = main_pump      OF fill_system_object
END IF

enable  active_pump
disable backup_pump
```

Figure 4.9 - Operator Selection of Pump

The fill_system_object contains all the objects of the LOX fill system. The operation 'main_pump OF fill_system_object' returns the object name of the main_pump system. The main_pump system contains all of the objects used to pump LOX with the main pump (for a better description of the fill_system_object see the next section). The 'LET active_pump =' assigns the active_pump variable the name of the appropriate pumping system object name. Again, it is important to remember that by using objects, the details of the pump object are hidden from the programmer of this high level control procedure. Once the active_pump and backup_pump variables are assigned the names of the active and backup objects, then an enable action is sent to activate up the pump to be used for this fill operation, and a disable action is sent to deactivate the backup pump as per the operator's wishes.

The rest of the SSLO2 code repeats elements that have been demonstrated already as shown in Figure 4.10. First, a setup and instrumentation status check is performed by sending the check_status action to the fill_system_object. The 'WITH' clause directs output of any anomalies found in the status check to the operator. Once the operator has been informed of the status, a wait suspends execution until the operator allows the procedure to continue. This type of operator involvement is common throughout the GOAL code we reviewed.

```
/* Setup and instrumentation status check (120 GOAL statements) */
check_status fill_system_object WITH output_to = operator_object

query operator_object WITH output  =
   "Instrument check complete, holding for pump start",
                            options = {'OK'}

SET speed OF active_pump TO 1000 RPM
start active_pump
WAIT startup_time OF active_pump
      OR UNTIL command_completed OF active_pump

query operator_object WITH output  =
   "Pump started, holding for chilldown",
                            options = {'OK'}
```

Figure 4.10 - Operational Statements

When execution continues, we send an action to set the speed of the pump to 1000 RPM. All interested objects that are part of the active pump will receive the action and operate on the action when the pump is started. The next statement sends a start action to the pump. The procedure then waits until the pump has started or the start has timed out (the error handling for the later case was not added). Again, the operator is informed of the pump start status, and the procedure waits for the operator to issue a continue command.

The SSLO2 procedure continues by executing each operation from the first chilldown to the final fill in the same manner as shown above: send the action, wait for results, and notify the operator. These operations are not shown in a figure. The organization of the procedure parallels the organization seen in the GOAL code. It should be noted that the organization of the procedure assumes that UIL is primarily event driven. If you review the GOAL code, you will find that it is also interrupt (event) driven. The last part of the GOAL procedure sets up the interrupt handlers to monitor the full status and transfers to a replenish procedure (maintain level of the tank to 100% full). Figure 4.11 shows the UIL version sending the action activate_topping to perform the same function.

```
write operator_object WITH output =
   "100% fill complete, transferring to topping"

activate_topping fill_system_object

END  /* of SSL02 */
```

Figure 4.11 - Transition of Control

While this top level procedure is somewhat simplified, the view that the top level routine is only responsible for high level functions (like check subsystem, start pump, etc) is the correct one. The elegance of the top level procedure depends on the forethought that goes into the object and action design. In the next three sections, a hypothetical design for these objects and actions will be presented. The design will begin by describing the basic object classes needed, then these object classes will be used to create more complex object classes, and finally, the objects themselves will be created.

4.3.2. Basic Object Classes

In this section, the UIL version of the GOAL open valve procedure GHK1F will be discussed. The UIL approach to the open valve function will be implemented as an object class. An object instantiated from this class will have an action that can open a valve. When this action is sent to the valve object, the valve will be opened. While the GHK1F procedure actually is used to open a valve for liquid hydrogen, the author used it as a model for a typical valve in the SSLO2 system as well which handles liquid oxygen.

As in most object based languages, UIL objects will have some default attributes and actions that the system will define automatically. In the case of UIL, the basic_object_class shown in Figure 4.12 contains examples of eight actions that each object should support. The 'new' action is called just after the system creates a new object for initialization of attribute data, similarly. The 'destroy' action is called to perform any cleanup functions just before the system removes an object from existence. The 'print', 'describe', 'list_actions', and 'is_action' actions provide either the user or a running procedure with information about the object. Finally, the 'unhandled_action' is called whenever an unknown action is received by the object. The unhandled_action procedure may handle the unknown action directly or, more conveniently, wish to raise an exception in a manner similar to ADA. Exceptions of this kind are not handled in the USEWG version of UIL. See Section 5.1.6 for further discussion on exceptions.

```
DEFINE CLASS basic_object_class

/*  Actions for the basic object will not be described.
    They perform at least the following functions:

 describe          - Types information about the object to the
                     operator.
 destroy           - Called when object is destroyed.
 is_action         - Returns true if this is an action of this
                     object.
 list_actions      - Returns the list of actions that the object
                     can handle.
 new               - New actions for object when created.
 null              - Action that does nothing.
 print             - Prints information about the object to an
                     I/O stream.
 unhandled_action - Handles undefined action requests. */

END DEFINE CLASS basic_object_class
```

Figure 4.12 - Basic Object Class

These actions provide the basic capabilities that most objects will be built upon. When a new subclass is built from the basic_object_class, all of the basic_object_class actions will be 'inherited' by the new class and are available for the new class. In many cases, an object subclass will override some (or all) of these actions by new actions that are tailored to the subclass.

Figure 4.13 shows the declaration portion of a new subclass being defined, called the pneumatic_valve_class. The 'IS basic_object_class' clause defines the class that the pneumatic_valve_class will inherit actions and attributes from, in this case, the basic_object_class. The 'CONSTANT' section defines attributes that should not be changed by an operating procedure (as we will see later, they can be overridden only by subclass definitions). The 'ACTION' declaration provides a mapping from the external action names to the internal procedures that will be called if the action is sent to an object of this subclass. The 'PARAMETER' section defines attributes that are accessed or changed by either the procedures defined in this subclass or by a procedure using an object created in this subclass. Following these declarations, the action procedure definitions occur. Several of those are shown in the next few figures.

```
DEFINE CLASS pneumatic_valve_class IS basic_object_class
   CONSTANT long_name = "pneumatic valve"
   CONSTANT type = "basic class"
   CONSTANT initial_component_motion = 1 SECOND .. 2 SECONDS
   CONSTANT home_time = 5 SECONDS .. 6 SECONDS

   ACTION close       IS close_procedure
   ACTION destroy     IS destroy_procedure
   ACTION disable     IS disable_procedure
   ACTION dynamics    IS dynamics_procedure
   ACTION enable      IS enable_procedure
   ACTION initial     IS initial_procedure
   ACTION new         IS new_procedure
   ACTION open        IS open_procedure
   ACTION revert      IS revert_procedure
   ACTION stop        IS stop_procedure
   ACTION warning     IS warning_procedure

   PARAMETER {input, output} IS basic_object_class
   PARAMETER {last_state, desired_state, current_state}
         IS ENUMERATED WITH VALUES = {'open', 'closed', 'unknown'}
   PARAMETER is_opened IS ENUMERATED WITH VALUES =
                            {'opened', 'not opened'}
   PARAMETER is_closed IS ENUMERATED WITH VALUES =
                            {'closed', 'not closed'}
   PARAMETER is_stopped IS ENUMERATED WITH VALUES =
                            {'stopped', 'not stopped'}
   PARAMETER transition_status IS ENUMERATED WITH VALUES =
       {'waiting for component motion', 'completed',
        'waiting for component home'}
   PARAMETER {open_status_bypassed, close_status_bypassed,

       command_status_bypassed}           IS shared_memory_class
   PARAMETER {open_status, closed_status} IS readable_hw_port_class
   PARAMETER command_status     IS readwriteable_hw_port_class
```

Figure 4.13 - Pneumatic Valve Class

The valve that this object subclass controls is of the same design as the valve shown in Figure 2.1. If you recall, it was designed such that it contains two indicators: one can detect when the valve has been completely opened, and the other detects when the valve has been completely closed. If the valve is in transition from open to close, neither of the two indicators is activated. The actions defined in this subclass provide the functionality to control a hardware valve of this type. Figure 4.14 shows the procedure defined for this subclass that overrides the definition of the 'new' action inherited from the basic_object_class. It initializes all attributes to their initial values.

```
DEFINE ACTION new_procedure IS
   /* executed only at instantiation of object */
   LET open_status_bypassed    = 'off'
   LET close_status_bypassed   = 'off'
   LET command_status_bypassed = 'off'
   ASK SELF TO warning_procedure WITH new_level = 'disable all'
   LET current_state = 'unknown'
   LET desired_state = 'unknown'
   LET last_state    = 'unknown'
   LET transition_status = 'completed'
   LET is_closed     = 'unknown'
   LET is_opened     = 'unknown'
   LET is_stopped    = 'not stopped'
END DEFINE ACTION new_procedure
```

Figure 4.14 - Pneumatic Valve New Action

A more interesting action procedure is the open_procedure that will be executed when an object receives an 'open' action. Shown in Figure 4.15, this procedure performs the necessary steps to open a valve. It parallels the design of the GKH1F GOAL open procedure discussed earlier. One of the most interesting statements in the figure is 'ASK SELF TO'. The word 'SELF' is a keyword that says to use the object, ITSELF, as the argument. Therefore, this statement sends the warning action to itself with the argument 'new_level = 'allow to open''. The UIL executor translates this into a call to the warning_procedure(new_level = 'allow to open'). The warning procedure sets up all event handlers to allow the open to begin without alerting the operator that an error has happened.

```
DEFINE ACTION open_procedure ( fast_mode ) IS
   PARAMETER fast_mode IS ENUMERATED WITH VALUES =
                              {'stopped', 'not stopped'}
   PARAMETER temp_hardware_state IS ENUMERATED
         WITH VALUES = { 'opened', 'likely opened', 'closed',
                'likely closed', 'bypassed', 'maybe open',
                'maybe closed' , 'conflicting', 'in transition' }

   LET temp_hardware_state =
      report_hardware_state_procedure

 -several statements omitted from the example at this point-

   LET desired_state = 'open'
   ASK SELF TO warning  WITH new_level = 'allow to open'

   LET transition_status = 'waiting for component motion'
   LET command_status = 'open'

   WAIT UPPER(initial_component_motion)
        OR UNTIL close_status = 'not closed'

 -several statements omitted from the example at this point-

   LET last_state = current_state
   LET current_state = 'open'
   LET transition_status = 'completed'
   RETURN 'ok'
END DEFINE ACTION open_procedure
```

Figure 4.15 - Pneumatic Valve Open Action

The hardware is then sent the command to open by setting the command_status to 'open'. This assignment changes the value of a hardware register that controls power to the solenoid that opens the valve. Remember from Figure 4.13 that the command_status was defined to be a readwriteable hardware port. Another interesting item in Figure 4.15 is the UPPER() operator. UPPER is used to get the upper value of the range indicated by the constant initial_component_motion.

It turns out that the control of this valve is very complicated. The combination of having both an 'open' and a 'close' status and bypass bits, creates a very complex problem. The action procedure get_hardware_state_procedure was used to determine that status of the hardware without any prior knowledge. Those interested in the author's solution to this problem should review the complete procedure listing in Appendix D.

4.3.3. Building other Object Classes

In this section, the instantiation of objects that represents the hardware controlled by SSLO2 will be described. The objects will be described from the lowest level first, for example, the valves and pumps. Once the low level objects are instantiated, then subsystem objects will be instantiated that contain the low level objects. Finally, the entire SSLO2 system will be described as a collection of subsystem objects.

In the last section, we discussed a valve subclass. By reviewing the code given in Appendix D, it can be found that the class requires an extensive definition. The next definition will reveal one of the power aspects of object based systems and of inheritance. In Figure 4.16, we define a new subclass of a specific valve of type a776 (the name was arbitrary). The a776 valve is slightly different in size, otherwise it is about the same as the previously discussed generic valve. It is a simple matter to create a new subclass for this a776 valve that inherits all the actions and attributes of the generic valve subclass. The 'REPLACE' keyword overrides the constants in the generic valve subclass. In just 8 lines, we created a new class!

```
DEFINE CLASS pneumatic_a776_valve_class IS pneumatic_valve_class
   REPLACE  long_name = "model A776 pneumatic valve"
   REPLACE  initial_component_motion = 2 SECONDS .. 6 SECONDS
   REPLACE  home_time = 18 SECONDS .. 20 SECONDS
   REPLACE  maximum_flow_rate = 35 CFM
   REPLACE  diameter = 45 CM

/* all other actions and object data inherited from
   pneumatic_valve_class */

END DEFINE CLASS pneumatic_valve_class
```

Figure 4.16 - Pneumatic A776 Valve Class

Figure 4.17 shows a different kind of an example object subclass. In this case, the subclass defined refers to a characteristic device rather than an actual hardware device. This 'precautions' subclass provides informations and actions that are related to hardware that handles LOX. Whereas the valve class grouped the functionality by hardware device, the 'precautions' class groups the functionality by the type of material that is being handled. As we will see in the next section, these two subclasses can be used to form a new subclass of object that is both a valve and handles LOX. This is possible through a mechanism called multiple inheritance.

```
DEFINE CLASS liquid_oxygen_precautions_class IS basic_object_class
   CONSTANT long_name = "oxygen precautions and alerts"
   CONSTANT type = "precautions class"
   CONSTANT boiling_point = -182.962 DEGC
   CONSTANT safe_childown_temp = -186.0 DEGC

   ACTION dynamics   IS dynamics_procedure
   ACTION new        IS new_procedure
   ACTION warning    IS warning_procedure

-several statements omitted from the example at this point-

END DEFINE CLASS liquid_oxygen_precautions_class
```

Figure 4.17 - Liquid Oxygen Precautions Class

4.3.4. Building the Objects Database

Using the classes defined in the last section, the last task left is to instantiate the objects that form a computer description of the hardware that SSLO2 controls. This can be thought of as creating a hardware schematic built with the objects in the computer's memory. The object's inputs and outputs are connected to the inputs and outputs of other objects. For example, a tank is connected to a valve, the valve to a pump, the pump to a transfer line, and so on. This section describes the instantiation of the objects themselves. The complete definition of subclasses and of the object instantiations are in Appendix D.

The general approach taken will be to instantiate the low level objects, like pumps and valves, first, to aggregate these low level objects into larger subsystems, and finally, to aggregate subsystems into a complete system. Figure 4.18 instantiates a pump object called pump_p128_object. The 'DEFINE OBJECT' statement creates an object in computer memory. The pump is instantiated from three different object parent classes: a pump_class, a hwsubsystem_class, and a precautions_class. These classes contain many different attributes and actions. The object instantiated will inherit the attributes and actions from all the classes. If two parent classes have definitions for the same attribute or action, then the attribute or action definition from the latter class in the parent class list will override the previous definition. The pump_p128_object inherits the actions needed to run the pump from the plus45_cm_pump_class. It inherits the actions to support its use as a part of a hwsubsystem from the sslo2_pump_hwsubsystem_class, and it inherits the warnings and safety actions from the liquid_oxygen_precautions_class.

```
DEFINE OBJECT pump_p128_object IS {plus45_cm_pump_class,
                                   sslo2_pump_hwsubsystem_class,
                                   liquid_oxygen_precautions_class}
   REPLACE  long_name = "Main fill pump"
   CONSTANT input     = lox_storage_tank_object
   CONSTANT output    = valve_p128_object
   CONSTANT control   = clutch_p128_object
   CONSTANT temp      = t7_object
   CONSTANT temp_byp  = byp_t7_object
END DEFINE OBJECT pump_p128_object
```

Figure 4.18 - Pump P128 Object

The constants defined in the pump object point to other objects that were instantiated earlier (not shown here). The input to the pump is LOX from the LOX storage tank object, and the output from the pump is LOX to a valve object. The clutch on the pump is controlled by an object that knows how to control clutches. Another object monitors the temperature, and yet another object provides the bypass flag. This type of object aggregation is typical in object based systems.

Continuing with the aggregation from smaller to larger subsystems, Figure 4.19 shows how the pump is associated with many other objects (the other objects have been instantiated in a similar way to the pump object). The UIL 'DEFINE OBJECT' statement causes the instantiation of an object of the hwsubsystem_class. This class serves as a container for subsystems. Some of the functions that this subclass might support is input/output to the operator or status checking. In this case, the object contains all the components of a pumping subsystem including temperature monitors, rate monitors, a motor, a clutch, a pump, a large valve, and a cooling subsystem pump.

```
DEFINE OBJECT sslo2_pump_hwsubsystem_object IS hwsubsystem_class
   REPLACE  long_name = "Primary pump128 hwsubsystem"
   CONSTANT contains  = {t1_object, t2_object, t3_object,
                         t4_object, t5_object, t6_object,
                         t7_object, t8_object, r1_object,
                         motor_p128_object, clutch_p128_object,
                         h2o_pump_p128_object, pump_p128_object,
                         valve_p128_object}
END DEFINE OBJECT sslo2_pump_hwsubsystem_object
```

Figure 4.19 - Pump Hwsubsystem Object

The pump subsystem is now capable of being used to pump LOX into the Shuttle. The operator could send an open start_pump to the subsystem, and the different objects would behave in the desired way as described in their internal action procedures. For example, the clutch would disengage, the power to the motor would then be started, the temperature sensors for the pump would be set to interrupt the pumping activity if the pump begins to overheat, and other objects would just sit tight.

Since the alternate pump subsystem has the exact same structure, the object instantiated in Figure 4.20 is the same as the object instantiated in the last figure. The 'DEFINE OBJECT' creates the new object using the old object as a guide. The define object specification would override many specific values, such as hardware port numbers, that are different.

```
DEFINE OBJECT sslo2_alt_pump_hwsubsystem_object
                    IS sslo2_pump_hwsubsystem_object
   /* redefine all the instantiation specific information
      (i.e., ports, connections, ...) of the
      sslo2_pump_hwsubsystem_class */
END DEFINE OBJECT sslo2_alt_pump_hwsubsystem_object
```

Figure 4.20 - Alt Pump Hwsubsystem Object

At this point, all the major objects have been instantiated and are placed in a higher level aggregation called the fill_system_object. Figure 4.21 shows the instantiation of this object. This object is used by the SSLO2 top level procedure. Again, the hwsubsystem_class of object serves as a container for major subsystems, that is, the primary and secondary pump subsystems and the LOX storage tank. The 'CONSTANT' declarations identify attributes that will hold the object names and other values of importance.

```
DEFINE OBJECT fill_system_object IS hwsubsystem_class
   REPLACE  long_name = "ET LOX loading and monitoring hwsubsystem"
   CONSTANT contains         = {sslo2_pump_hwsubsystem_object,
                               sslo2_alt_pump_hwsubsystem_object,
                               sslo2_lox_tank_object}
   CONSTANT main_pump        = sslo2_pump_hwsubsystem_object
   CONSTANT alternate_pump   = sslo2_alt_pump_hwsubsystem_object

/* fill_system specific constants */
   CONSTANT slow_fill_time_limit     = 11 MINUTES
   CONSTANT fast_fill_time_limit     = 30 MINUTES
   CONSTANT complete_fill_time_limit = 10 MINUTES
END DEFINE OBJECT fill_system_object
```

Figure 4.21 - Fill System Object

In this thesis, all the object classes and objects were defined in UIL, and were instantiated (from the point of view of the parser) at syntax check time. The actual object instantiations would, most likely, be performed by NASA and would be carefully controlled. It is also presumed that many tools will be created to allow the graphical and textual editing of objects; therefore, it is unlikely that the SSLO2 subsystem would be instantiated by the hard coded list that appears in Appendix D. The reader will also notice that the SSLO2 control procedure was written using the syntax designed for objects that were not created by UIL. This was done for clarity only. The action/object statements in SSLO2 would appear exactly as shown in Figure 3.15 had they been instantiated from UIL objects.

5. UIL EXTENSIONS

The USEWG version of UIL does not fully support the needed functionally of at least one of the languages it was intended to replace (GOAL). In this chapter, several recommended changes to the syntax of UIL are summarized, and issues concerning the implementation of UIL are discussed. However, some of the issues that are discussed are far from resolved, and their final resolution is beyond the scope of this thesis.

5.1. Recommended UIL Extensions

This section summarizes several major changes to the syntax of UIL to support the porting of GOAL code, such as the SSLO2 example, into UIL. Some of the changes have been discussed before and are repeated here only for the purpose of completeness.

5.1.1. Support UIL Defined Classes

Support should be added in UIL for the creation new classes and objects. The current USEWG version of UIL is an object manipulation language, not an object based language. The difference is that UIL can send actions to objects and gain access to attributes in objects; however, UIL cannot define new complex object classes or objects. The complex objects used in the USEWG version of UIL are created by ADA programs, and they are made available to the UIL executor through the operating system. The additions to the UIL syntax shown in Figure 5.1 allow UIL to support the ability to create complex classes and objects with UIL definable actions and attributes. The detailed syntax of these object and class declarations was discussed in Section 3.10.

```
declaration_extentions = ( big_object | big_class ) .

/* create an action as a sequence of UIL statements */

uil_action =
   'DEFINE' 'ACTION' internal_name
      [ '(' parameter_name_list ')' ] 'IS'
      { $ sequence_of_statements }
   'END' 'DEFINE' 'ACTION' [ internal_name ] .

/* external_name is a action name that will be used to send to an
   object - internal_name is the procedure name as defined in the
   object */

action_declaration = 'ACTION' external_name 'IS'
                          ( internal_name | 'NULL' )

big_class =
   'DEFINE' 'CLASS' class_list [ 'IS' class_list ]
      { $ ( declaration | action_declaration ) }
      { $ uil_action }
   'END' 'DEFINE' 'CLASS' [ class_list ] .

big_object =
   'DEFINE' 'OBJECT' object_list 'IS' class_list
      { $ ( declaration | action_declaration ) }
      { $ uil_action }
   'END' 'DEFINE' 'OBJECT' [ object_list ] .
```

Figure 5.1 - Object Definition Extensions

5.1.2. Support Generic Action Sending

UIL should provide a method for sending an action to an object for new object classes created at run-time. Normally, the operating system provides the UIL executor with the actions and objects that the UIL procedure will be able to use. The object and action names are required before the UIL program can be compiled because the action and object name occur in the code without any special syntactic clues, for example, the UIL statement to open a valve could look like 'open valve'. This syntax presents a problem if a UIL program uses the object definition extensions described in the last section. To eliminate this problem, a generic send action syntax is proposed in Figure 5.2 to allow an action to be sent to an object that is not defined in a dictionary. Example generic send action commands would be 'ASK valve TO open' or 'SEND open TO valve'. The syntax of generic send action was discussed in detail in Section 3.10.

```
action_statement = send_action .

send_action = ( 'SEND' an_action 'TO' an_object) |
              ( 'ASK'  an_object 'TO' an_action) .

an_action = ( action_dictionary_entry | action ) [ with_clause ] .
an_object =    'SELF' | object_name | object_dictionary_entry  .
```

Figure 5.2 - Action Definition Extensions

Both forms of the send action shown above (with SEND or ASK) are functionally equivalent. The two forms are provided because sometimes it reads better to 'ASK' an object to do an action, and other times if reads better to 'SEND' an action to an object. These more formal forms allow any message and object to be used without conflicts or parsing problems.

5.1.3. Support Lists

UIL should provide better support of the list data type within the language, and not as an object class. There are many uses of lists that are difficult to handle with fixed sized arrays or other data types. The GOAL code made extensive use of lists to keep the code compact and readable. UIL should also provide an unambiguous syntax to clearly indicate that the data is in a list. UIL support of lists should be allowed in almost any expression. Lists are too important a data type to be an 'added-in' feature of the language. They must be fast in execution and easy to use. Figure 5.3 defines the syntax for two kinds of lists and provided several examples of lists in use.

```
expression_list =
   expression | ( '{' expression { $ ',' expression } '}' ) .

name_list =
   name       | ( '{' name       { $ ',' name       } '}' ) .

                LET {a, b, c} =  FALSE
                LET {a, b, c} = {TRUE, FALSE, TRUE}
                LET a         = {TRUE, FALSE, b, c}
                LET a         =  TRUE
                IF  {a, b, c} = {TRUE, FALSE, TRUE} THEN ...
                IF  {a, b, c} =  TRUE               THEN ...
```

Figure 5.3 - List Extensions

5.1.4. Support Arrays and Structures

UIL should support arrays and data structures as basic data types for the same reasons as described in the last section on lists. The USEWG version of UIL does not support arrays and data structures. These data types should be similar to those found in other modern conventional languages, for example, ADA or C. Objects do NOT replace all types of data structures within a program. Conventional data structures perform many of the desired jobs in the best possible manner. It is not acceptable to say that an object class can be created for any type of data structure that is needed. In many cases, the overhead that an object creates will be unacceptable.

5.1.5. Support Generic Pathnames

UIL should be designed to be portable from the beginning. Non-portable aspects should not be introduced into the language. The USEWG version of UIL is not portable from at least one aspect, the pathnames. There is no excuse for creating a modern language that is not portable. Generic pathnames should be used in UIL. The run-time system or operating system can translate the generic pathnames into the host computers preferred format. It is easy to create a generic filename format (or seek out some standard that is already in use). A standard filename format will greatly simplify porting efforts. Table 5.1 provides a trivial example of some filename translations.

Table 5.1 - Example of Generic File Names

System Dependent	UIL System Independent
Vax::Dev:[a.b.c]file.ext	Vax;;;Dev;;a;b;c;file.ext
/user/orbit/datafile	user;orbit;datafile

5.1.6. Support Event Handlers

UIL should provide support to define, attach, detach, enable, disable, check, signal, change levels, and override event handlers from within a UIL procedure. Up to 25% of the code in the GOAL procedures we reviewed were related to handling events. UIL procedures should be able to manage events under statement control. There is a temptation to implement event_handlers as standard objects; however, the execution time constraints associated with events will, almost certainly, require events to be implemented as basic language features. The current UIL specification only allows events to be used from outside UIL procedures, and does not support multi-priority events. Figure 5.4 shows some of the syntactical extensions that UIL will require to support events. These event handler extensions follow directly from our review of the GOAL code.

```
event_attach_statement =
   'EVENT' 'HANDLER' 'FOR' event_name
       'IS' ( sequence_of_statements | 'NULL' )
       [ 'LEVEL' interrupt_level ]
   'END' [ event_name ] .

event_enable_statement =
   ( 'ENABLE' | 'DISABLE' | 'SIGNAL' | 'CHECK' | 'QUERY' |
      ( 'CHANGE' [ 'PENDING' ] 'LEVEL' ) )  event_name .

event_enable_level_statement =
   ( 'ENABLE' | 'DISABLE' | 'CHECK' | 'QUERY' )  event_level .

exception_statement = 'RAISE' event_name .
```

Figure 5.4 - Event Handler Extensions

The 'RAISE' statement has been added to allow an event to be raised from within a UIL procedure. The USEWG version of UIL defines exception handling for only expression evaluation. There are many cases of procedure error handling that can best be supported by an event style of exception handler. The addition of the raise statement provides the maximum flexibility in handling errors or exceptional conditions by allowing the programmer to use either a conventional error handling procedure or by using event handlers.

5.1.7. Support Error Handling for Timed Sections

UIL should support simpler handling of timed sections of code by adding additional clauses to the wait_statement that would make timed operations easy to code. There are many operations that involve time, in fact, UIL defines special data types for times and dates. To simplify the programming involved for timed functions, UIL should support additional clauses for the wait statement like those shown in Figure 5.5. There are two new clauses added to allow a sequence of statements to be executed when a time out condition occurs. The raise statement described in Section 5.1.6 provides a convenient method of handling time out exceptions. These additions will simplify the error handling of timed operations.

```
wait_statement = 'WAIT'
       [ ( 'UNTIL' logical_expression ) |
         ( timing_simple_expression   ) |
         ( timing_simple_expression
              [ 'OR' 'UNTIL' logical_expression ] ) ]
/*new*/ [ 'ON' 'TIMEOUT' sequence_of_statements 'END' 'TIMEOUT' ]
/*new*/ [ 'ON' 'UNTIL'   sequence_of_statements 'END' 'UNTIL'   ] .
```

Figure 5.5 - Timed Section Extensions

5.2. Binding of UIL objects

The term binding refers to the time when UIL connects the actions to the objects. If binding occurs 'early' at compile time, the code executes much faster; however, the program flexibility is restricted by the fact that all object structures are fixed at compile time. ADA is an example of a language that does early binding. On the other hand, if binding occurs 'late' at execution time, the object structures can be changed at run-time; however, there is a significant run-time overhead attached to this enhanced flexibility. Smalltalk-80 is an example of a language with late binding.

To crystalize the concept of binding, let us apply binding to a simple hardware subsystem example. Given that we have a subsystem with two valves and one pump: what does early binding or late binding imply about the subsystem? In an early binding system, the subsystem would be compiled into code in one piece, and action procedure calls for the valves and the pump are fixed into code. When an action is sent to the subsystem, these procedure calls will be executed, and the results are returned. In a late binding system, the subsystem objects could be compiled into code together or separately or even created by a UIL procedure at run-time. The action procedures cannot be determined until the action is sent. When an action is sent to the subsystem, the UIL executor will dispatch the action to the action procedures that are, at this instant in time, attached to each of the three objects.

The preferred binding method of UIL objects and subsystems will depend on the context of use, the implementation of the UIL compiler, and the Space Station operating environment. There are at least three separate cases to be considered: fixed hardware subsystems, changing hardware subsystems, and user driven applications.

At a first glance, a fixed hardware subsystem would seem to be the only application of UIL procedures. Most subsystems aboard Space Station would fall into this category (at least, this appears to be the case at a first glance). As an example, let us call the Shuttle a hardware subsystem. The Shuttle is a very complex system that could be compiled into a Shuttle object. The compiler (after spending three days optimizing the code) generates a very fast executing Shuttle object. Now, we send the Shuttle object the 'take off' action, and it blasts into the sky as planned, and successfully goes into orbit. Everything is fine until a flight computer fails and changes the Shuttle object configuration. Recompiling this object is out of the question.

This example may be a bit extreme! Let us redefine the Shuttle object to refer to a collection of LRUs (Line Replaceable Units). A line replaceable unit is a small subsystem that cannot be changed internally; however, they can be removed and replaced by other LRUs in flight. The compile time binding is very useful for a LRU as the LRU object would execute very fast and could be substituted for other LRUs in the case of a hardware failure. The connections between the LRUs could be either special purpose code written into the UIL procedure or late binding of just the LRUs. If there is a hardware change in operation, such as repairing or changing flight components within a LRU, it would be likely that the LRU hardware subsystem will be 'recompiled'. It is even possible that a different subsystem object would be compiled for each LRU serial number due to slight differences in operating characteristics.

The concept of a system with more than one LRU is an example of a changing hardware subsystem. There must be a method of 'connecting' hardware subsystems without recompiling the objects. Another extreme example is as follows: the Shuttle object should, without being recompiled, be able to be placed on either pad A or pad B (pad A and pad B are the two, slightly different, Shuttle launch sites at KSC). The differences between the two pads are currently hard-coded into operational GOAL procedures. Another example would be the replacement of a LRU in flight. Each functionally different LRU should have its own precompiled UIL hardware subsystem object that can be 'plugged' into the UIL subsystem controller software or a simulator. One last example follows: the status output and control input for a subsystem should be bound to a console or controller at run-time as desired, and changed or moved as needed.

Whereas hardware systems do not change frequently, the user driven applications of UIL must be free to instantiate and destroy objects and object subsystems at run-time. For example, windowing systems and most other operator directed interactions are driven by the user, and it is simply not feasible to plan ahead how many windows the user may be using at any one time. In these cases, early binding cannot be used.

The conclusion to the discussion is this: It seems that UIL should support both early binding and late binding. The Space Station will be composed of early binding LRUs that are compiled, and late binding slots that LRUs will be plugged into. When UIL is used to support dynamic operations like user interfaces, the binding will be, in all likelihood, late.

5.3. Object Based Hardware Subsystem Descriptions

Related to the binding issue discussed in the last section, there are some questions on how to build and maintain the subsystem objects. In the SSLO2 example, a subsystem object was built of smaller objects that represent the valves and pumps. What was not discussed in detail is how the subsystem object was built. The comment in Chapter 4 was that all the objects were instantiated before they were combined into a subsystem. There are problems with this: what about forward references? If object A points to object B and object B points to object A (e.g., when the output of A is connected to the input of B and vice versa), then how do you instantiate the 'first' one? At least two solutions are possible: multi-pass object instantiation, and generation of the object connections after all instantiations have been completed.

In the multi-pass instantiation method, all the objects would be instantiated by the compiler, and the attribute values would be filled in after the first pass. In this way, the forward references would not be a problem. Using an object connection scheme after the instantiation of objects could be more flexible, for example, make objects A, B, and only then add the connections between them. This could replace the multi-pass system; however, this implies that there is some type of special 'code' that would follow the instantiation of objects to connect them (via attribute values). This 'code' fills in attribute values, but it does not generate executable code. Neither solution is ideal.

5.4. Multitasking and Testability in UIL

There are many issues, such as testability and multi-tasking, that were not covered in this thesis; however, there are several observations about the GOAL code that may effect the implementation of UIL. The current design of UIL makes no mention of multi-tasking, or how events would be handled. Observations of the GOAL code we reviewed and the nature of controlling multiple devices suggest a multi-threaded procedure execution. The term multi-threaded is used here to refer to several parallel execution paths through the code. The multiple threads could be implemented either via multi-tasking or by using event handlers or both. The object design concepts of UIL may already come as a shock for the current users of GOAL. To say further that UIL will support multiple threads of execution may be even more disturbing from a testability point of view.

On the other hand, one has to believe that UIL can be designed to be a more productive language than GOAL. The GOAL language does not even provide the most basic tools for the programmer. The current approach to developing procedures in GOAL for the Space Shuttle is best described as event driven. The code is very 'flat' and never nests more than a level or two of 'if ... then ... else'. The GOAL version SSLO2 sets up over 290 interrupt handlers. The language depends on Shuttle hardware to generate interrupts on every aspect of the control. In fact, all hardware devices have front end processors that receive and process these interrupts.

There are several reasons for the way SSLO2 was designed and coded with such a dependence on interrupt handlers:

(1) There is no CPU time spent 'polling' for external events.

(2) The high priority interrupts can be handled quickly by the front end processor, and can later be raised to the GOAL processor.

(3) Flat code is easy to time and test and likely to execute faster.

(4) The control flow is straightforward (almost a straight line).

UIL currently avoids the issue of multi-threaded execution. In SSLO2, there are many actions that go on simultaneously and that are executed by priority or in turn by interrupt handlers. The object based approach of UIL won't do much good with the SSLO2 example unless it is augmented by some mechanisms for multi-threaded execution. The question is whether these inherent capabilities should penetrate through to the user surface, for example, whether the user should be made aware of the multi-threaded execution. This thesis does not provide an answer to this difficult, yet important, question.

6. CONCLUSIONS

The development of the Space Station FREEDOM is an enormous effort, and the success of the project will depend on many interlocking pieces of technology. UIL is one of these pieces of technology. It provides users powerful access to Space Station resources without the need to write procedures in a conventional computer programming language. UIL is another step in the evolution of aerospace operations languages. UIL will provide the man-machine interface for most of the test and operation command procedures, development and testing of experiments on the ground, and the remote execution of these experiments.

The work completed for this thesis provides a concrete test case for UIL by porting an operational GOAL procedure called SSLO2 from the Space Shuttle launch system to UIL. The hardware controlled by this procedure should be representative of the types of applications that UIL will be used to control on the Space Station FREEDOM. By suppling a syntax parser program with the proposed UIL syntax, the translated UIL version of SSLO2 was verified to be syntactically correct. In the process of translating all of the functionality of SSLO2 into UIL, several inadequacies were discovered in UIL. The syntax of UIL was then enhanced to provide the functionality to support applications such as SSLO2.

The object oriented approach that was taken with UIL is based on the successful application of object oriented concepts to several other systems in operation today. The object oriented approach has several advantages that are of use to the Space Station program. The object oriented approach provides convenient packaging of code and data. It is comforting to deal with the control of hardware devices at the level of 'open valve' or 'start pump' where the details of how the valve opens is hidden from the user. For example, once the objects were properly defined, the actual high level control part of the SSLO2 procedure was short compared to the GOAL code. The powerful concept of class inheritance simplifies the creation of new object classes. The data driven nature of object based systems can be used to flexibly change hardware configurations without the need to recompile a control program.

A significant amount of time was spent analyzing the provided GOAL procedures. These procedures and the GOAL run-time environment provide a glimpse of the past and a warning for the future. The GOAL environment was primitive, by todays standards: huge unstructured procedures, special purpose hardware and software, and poor programming tools. This should not come as a shock for those involved, given that the initial design of the system dates back more than 15 years. It is likely that the current Space Shuttle launch system will continue to operate for another 5 years or more. The warning is this: the UIL language that is defined today will influence the implementation of Space Station applications and other NASA projects, such as the manned lunar base, for many years to come; thus, a short-sighted approach to design questions today will have detrimental effects that will hamper the future development of many projects, projects that are not even in the early design stages now.

In this thesis, several major enhancements to UIL were proposed. The three most important enhancements were: UIL support for event handlers, UIL support for data structures, and UIL support for creating complex classes and objects. The addition of these enhancements change the character of UIL from an object manipulation language to an object based language. The enhancements will profoundly change the future of UIL and expand the number of applications that will use UIL.

APPENDIX A

UIL SYNTAX DEFINITION

```
.OPT
    METASYMBOLS
      OR | ;

  SETOPTION
       INTERSECTION ;

  SYNDIA
       PRTPLOT 100 ,
       PRTBNF 70 ,
       PLTWIDTH 20.0 ,
       PLTCHWIDTH 0.25 ,
       PLOT DISPLAY ,
       LABELS ;

   REDEFINITIONS
     COMMENT  /* */ ;

$$$$$
```

```
/*****************************************************************
 *
 *          Space Station UIL (User Interface Language)
 *         LANGUAGE SYNTAX DEFINITION Version 1.1 - 6/5/89
 *
 *  In the following lines, an '\' is actually a '/' for comments
 *
 *  Modified original version of the UIL syntax for LL(1) parsibility:
 *     \*+ shows added info *\   \*- shows subtracted info *\
 *
 *  Modified for SSLO2 version:
 *     \**+ shows added info *\   \**- shows subtracted info *\
 *     \*#+ shows added info *\   \*#- shows subtracted info *\
 *     (Made all keywords uppercase for clarity)
 *    - Added return value for return statement:
 *    - Extended 'with_clause' to handle special cases in text
 *    - Allowed program control of event handers
 *    - Allowed function calls with zero arguments
 *    - Added support for unique form of a list i.e., {a,b,c}
 *    - Allowed use of a list for parameter name
 *    - Added REPLACE to allow redefine of a CONSTANT
 *    - Added example object/action dictionaries
 *    - Added example unit names
 *    - Added UIL definitions for complex objects
 *         (big_class, big_object, uil_action)
 *    - Added generic send action forms
 *
 *****************************************************************/

/* 3 language definition */
/* 3.1 about language definition */
/* 3.1 notation */
/* 3.2 character set -- see end of this file */

/* 3.3 lexical elements */
/* 3.3.1 words */
word = 'IDENT' /* letter { $ letter | digit } */ |
/**+ Allows words to be OBJECT names */ object_directory_entry .

/* 3.3.1.1 reserved words */
/* 3.3.1.2 words that indicate direction */
direction = 'ON' | 'OFF' | 'IN' | 'OUT' | 'UP' | 'DOWN' |
            'LEFT' | 'RIGHT' | 'FORWARD' | 'BACKWARD' |
            'CLOCKWISE' | 'CW' | 'COUNTERCLOCKWISE' | 'CCW' .
```

```
/* 3.3.2 numbers */
/* 3.3.2.1 integer numbers */
integer = 'UINTEGER' /* digit { $ [ '_' ] digit } */ .
/*+*/ based_integer = base extended_part .
/*+*/ extended_part =        '#' extended_digit
                { extended_digit | ( '_' extended_digit ) } '#' .
/*-   based_integer = base '#' extended_digit
                             { $ [ '_' ] extended_digit } '#'. */
base = integer .
extended_digit = digit | letter .

/* 3.3.2.2 real numbers */
/*+*/ real = integer real_part .
/*+*/ real_part =     '.' integer [ exponent ] .
/*-   real = integer '.' integer [ exponent ] . */
exponent = ( 'E' | 'e' ) [ '+' | '-' ] integer .

/* 3.3.3 dates and times */
date_time = date '-' time .
count_time = day '-' time .

date = year '/' ( ( month '/' day_of_month ) | day_of_year ) .
time = hours ':' minutes [ ':' seconds ] .
day = integer .

/*+*/ year = digit digit digit digit .
/*-   year = integer . */
month = integer .
day_of_month = integer .
day_of_year = integer .

hours = integer .
minutes = integer .
seconds = integer [ '.' integer ] .

/* 3.3.4 strings */
STRING = ( ''' ''' ) | ( '"' '"' ) . /* STRING predefined */
text_string = STRING .
pathname_string = STRING .

/* 3.3.5 special characters */
/* 3.3.6 separating lexical elements */
/* white space is spaces and/or tabs */

/* 3.4 basic syntactic components */
/* 3.4.1 names */
name = { word } .
```

```
/* 3.4.1.1 names of objects, object classes and attributes */
/*-   object_identifier = object_name | pathname_literal . */
/*+*/ object_identifier =
        ( name [ ':' pathname_string ] ) | pathname_string .
/*+*/ attribute_identifier =
        attribute_name 'OF' object_identifier .

/* 3.4.2 literals */
/* 3.4.2.1 numeric literals */
/*+*/ numeric_literal = integer [ extended_part | real_part ] .
/*-   numeric_literal = integer_literal | real_literal . */
/*+*/ integer_literal = integer [ extended_part ] .
/*-   integer_literal = integer | based_integer . */
      real_literal    = real .

/* 3.4.2.2 time literals */
date_time_literal = date_time time_zone_name .

/* 3.4.2.3 string literals */
text_literal = text_string .
pathname_literal = [ operating_system_name ':' ] pathname_string .

/*-   string_literal = text_literal | pathname_literal . */
/*+*/ string_literal = [ operating_system_name ':' ] STRING .

/* 3.4.2.4 enumeration literals */
extended_name = { $ word } direction { $ ( word | direction ) } .
/*-   enumeration_literal = name | extended_name . */
/*+*/ enumeration_literal =
        ( { word } [ direction { $ ( word | direction ) } ] ) |
        (            direction { $ ( word | direction ) }   ) .

/* 3.4.2.5 measurement literals */
measurement_literal = measurement { $ measurement } .
measurement = numeric_literal units .
units = units_name | ( '<<' units_expression '>>' ) .
units_expression = units_factor {$ multiplying_operator units_factor } .
units_factor =
   units_name [ exponentiation_operator units_exponent ] .
units_exponent = [ '+' | '-' ] integer_literal .

/* 3.4.3 lists */
list = expression { $ ',' expression } .
```

```
/* 3.4.4 expressions */
/*+*/
expression =
  ( ( simple_expression
          [ ( relational_operator simple_expression )  |
            ( range_test_operator range                 )  |
            ( range_operator simple_expression          )    ] ) |
    ( logical_not_operator simple_expression            )    )
  [ ( 'AND' relation { $ 'AND' relation } ) |
    ( 'OR'  relation { $ 'OR' relation }  )    ] .
/*-
expression =
        ( relation [ ( 'AND' relation { $ 'AND' relation } ) |
                     ( 'OR'  relation { $ 'OR' relation }  ) ] ) |
        range . */

relation =
       ( simple_expression
            [ ( relational_operator simple_expression ) |
              ( range_test_operator range ) ] )            |
       ( logical_not_operator simple_expression ) .

range = simple_expression range_operator simple_expression .

simple_expression = [ unary_adding_operator ]
     term { $ binary_adding_operator term } .
term = factor { $ multiplying_operator factor } .

factor = ( primary [ exponentiation_operator primary ] ) .

/*+*/
primary =
/*-int          numeric_literal |  measurement_literal | */
/*+int*/      ( numeric_literal
                  [ units { $ numeric_literal units } ] ) |

/* digit*/      date_time_literal |

/*-strg/name    string_literal | */
/*+strg/name*/  STRING |          /*part of, pathname in 'name' section*/

/*-name         enumeration_literal | object_name | function_call |
                attribute_identifier | indexed_object | */
/*+name*/
              ( { word }
                  [ ( ':' STRING ) |
                    ( 'OF' object_identifier ) |
                    ( '(' [ list ] ')' ) |
                    ( direction { $ ( word | direction ) } ) ] ) |

             (  direction { $ ( word | direction ) } )         |
/*list*/          ( '(' list ')' )  |
/**+ support unique form for a list */
                  ( '{' list '}' ) .

/*-
primary = numeric_literal | date_time_literal | string_literal |
          measurement_literal | enumeration_literal |
          object_name | attribute_identifier |
          indexed_object | function_call |
          ( '(' list ')' ) . */

/* 3.4.4.1 operators and their precedence */
logical_connecting_operator = 'AND' | 'OR' .
logical_not_operator =         'NOT' .
relational_operator =        ( 'IS' [ 'NOT' ]  ) | '=' | '/=' |
                               '<' | '<=' | '>' | '>=' .
range_test_operator =          'IS' [ 'NOT' ] 'WITHIN' .
range_operator =               '..' .
unary_adding_operator =        '+' | '-' .
binary_adding_operator =       '+' | '-' | '&' .
multiplying_operator =         '*' | '/' .
exponentiation_operator =      '**' .

/* 3.4.4.2 ranges */
/* 3.4.4.3 expression evaluation */
/* 3.4.4.4 objects names in expressions */
/* 3.4.4.5 functions */
function_call = function_name '(' [ argument_list ] ')' .
/*#+ added optionals around argument_list to allow functions
     to be called without arguments */

/* 3.4.4.6 indices */
indexed_object = object_name '(' index_list ')' .

/* 3.4.4.7 exceptions during expression evaluation */
/* 3.4.4.8 examples of expressions */

/* 3.5 statements */
sequence_of_statements = { statement } .
statement = declaration | command |
     assignment_statement | sequential_control_statement |
     conditional_control_statement | iterative_control_statement |
/**+ Allow program control of event handlers */ event_handler |
/*#+ Allow UIL definition of complex objects */ extensions .

/* 3.5.1 declarations */
declaration = subclass_declaration | parameter_declaration |
              constant_declaration |
/*-           object_declaration   | rename_declaration . */
/*+*/         object_rename_declaration .
```

```
/* 3.5.1.1 subclass declarations */
subclass_declaration = 'CLASS' object_name
                        'IS' object_name [ with_clause ] .
with_clause = 'WITH' assignment { $ ',' assignment } .
assignment = ( name '=' expression ) |
/**+  These added only for clarity */
/**+*/        ( 'DEFAULT' 'VALUE' '=' expression ) |
/**+*/        ( 'INITIAL' 'VALUE' '=' expression ) |
/**+*/        ( 'RANGE' '=' range ) |
/**+*/        ( 'MODE' '=' ( ( 'IN' [ 'OUT' ] ) | 'OUT' ) ) .

/* 3.5.1.2 object declarations */
object_declaration = 'OBJECT' object_name
                              'IS' class_name [ with_clause ] .
/***** See section 3.5.1.5 */

/* 3.5.1.3 parameter declarations */
parameter_declaration =
     'PARAMETER' parameter_name_list
         'IS' class_name [ with_clause ] .
/*#+ Allowed use of a list for parameter names - easier creation
     of parameters that are all the same class */

/* 3.5.1.4 constant declarations */
constant_declaration = 'CONSTANT' assignment |
/*#+ Added REPLACE to allow redefine of CONSTANT */
                       'REPLACE'  assignment .

/* 3.5.1.5 renaming objects */
rename_declaration = 'OBJECT' object_name 'RENAMES' object_name .
/*+*/ object_rename_declaration = 'OBJECT' object_name
           ( ( 'IS'      class_name [ with_clause ] ) |
             ( 'RENAMES' object_name )             ) .

/* 3.5.1.6 the scope of objects */
destroy_command = 'DESTROY' object_name_list .

/* 3.5.2 commands */
command = basic_command [ qualifier_group ] .

/* 3.5.2.1 basic commands */
basic_command =
   ( action ( commanded_object_list [ direction ] ) |
            ( direction commanded_object_list )       ) |
/**+ Added example object/action dictionaries */
   example_command .

commanded_object_list =  object_identifier_list [ 'OF'
                          object_identifier_list ] .
action = verb | ( 'START' verb ) | ( 'STOP' verb ) .
verb = word .
```

```
/* 3.5.2.2 qualified commands */
qualifier_group = qualifing_clause { $ ',' qualifing_clause } .
qualifing_clause = ( qualifier expression ) | with_clause .
qualifier = 'AFTER' | 'AT' | 'BEFORE' | 'BY' | 'EVERY' |
            'FROM' | 'INTO' | 'TO' | 'UNTIL' | 'WHERE' .

/* 3.5.3 assignment statement */
assignment_statement = 'LET' assignment .

/* 3.5.4 sequential control statement */
sequential_control_statement = null_statement | step_statement |
                   go_to_statement | wait_statement |
                   return_statement .

/* 3.5.4.1 null statement */
null_statement = 'NULL' .

/* 3.5.4.2 step statement */
step_statement = 'STEP' [ step_id ]
   'IS' sequence_of_statements 'END' 'STEP' .
step_id = name | integer [ name ] .

/* 3.5.4.3 go to statement */
go_to_statement = 'GO' 'TO' 'STEP' step_id .

/* 3.5.4.4 wait statement */
wait_statement = 'WAIT' [
   ( 'UNTIL' logical_expression ) |
   ( timing_simple_expression
      [ 'OR' 'UNTIL' logical_expression ] ) ]
/*new*/ [ 'ON' 'TIMEOUT' sequence_of_statements 'END' 'TIMEOUT' ]
/*new*/ [ 'ON' 'UNTIL'   sequence_of_statements 'END' 'UNTIL'   ] .

exception_statement = 'RAISE' event_name .

/* 3.5.4.5 return execution */
return_statement = 'RETURN' [ 'ALL' ] /**+*/ [ expression ] .

/* 3.5.5 conditional control statement */
conditional_control_statement = if_statement | verify_statement |
                   case_statement .

/* 3.5.5.1 if statement */
if_statement = 'IF' logical_expression 'THEN'
                sequence_of_statements
             { $ 'ELSE' 'IF' logical_expression 'THEN'
                sequence_of_statements }
             [ 'OTHERWISE'
                sequence_of_statements ]
             'END' 'IF' .
```

```
/* 3.5.5.2 verify statement */
verify_statement =
            'VERIFY' logical_expression
               [ 'WITHIN' timing_simple_expression ]
            [ 'THEN'
               sequence_of_statements ]
            [ 'OTHERWISE'
               sequence_of_statements ]
            'END' 'VERIFY' .

/* 3.5.5.3 case statement */
case_statement = 'CASE' expression
            { $ ( 'WHEN' logical_expression
                    'THEN' sequence_of_statements ) }
            [ 'OTHERWISE' sequence_of_statements ]
            'END' 'CASE' .

/* 3.5.6 iterative control statements */
iterative_control_statement = repeat_statement | while_statement |
            for_statement | exit_statement .

/* 3.5.6.1 repeat statement */
repeat_statement = 'REPEAT' sequence_of_statements 'END' 'REPEAT' .

/* 3.5.6.2 while statement */
while_statement = 'WHILE' logical_expression repeat_statement .

/* 3.5.6.3 for statement */
for_statement = 'FOR' index_object_name '=' list repeat_statement .

/* 3.5.6.4 exit statement */
exit_statement = 'EXIT' [ 'IF' logical_expression ] .

/* 3.6 environments, procedures and event handlers */
/* 3.6.1 environments */
/* 'INSTALL' environment [ 'ON' processor ]
                [ 'WITH' 'PRIORITY' '=>' expression ] */
/* 'REMOVE' environment */

/* 3.7 procedures */
procedure = 'PROCEDURE' procedure_name
                [ '(' parameter_name_list ')' ]
  'IS' sequence_of_statements 'END' [ procedure_name ] .

/* 3.7.2.1 executing a procedure */
/* 'EXECUTE' procedure_name [ 'IN' environment ]
        [ 'FROM' 'STEP' integer ]
        [ 'TO' 'STEP' integer ] [ 'WITH' parameter_list ] */

/* 3.7.2.2 terminating a procedure */
/* 'TERMINATE' procedure_name [ 'in' environment ] */
/* 3.7.2.3 suspending a procedure */
/* 'SUSPEND' procedure_name [ 'IN' environment ] */

/* 3.7.2.4 resuming execution of a suspended procedure */
/* 'RESUME' procedure_name [ 'IN' environment ] */

/* 3.7.2.4 redirect a procedure */
/* 'REDIRECT' procedure_name [ 'IN' environment ]
                     'TO' 'STEP' step_name */

/* 3.8 events and event handlers */
event_handler = 'EVENT' 'HANDLER' 'FOR' event_name
                'IS' sequence_of_statements
                'END' [ event_name ] .

/* 3.8.3.1 enabling and disabling events */
/* 'ENABLE' 'NOTIFICATION' 'OF' object_name */
/* 'DISABLE' 'NOTIFICATION' 'OF' object_name */

/* 3.8.3.2 signalling an event */
/* 'SIGNAL' object_name [ 'TO' list ] */
/* Primitive definitions (predefined) */

letter = 'LETTER' .
digit =  'DIGIT' .

/*
character = letter | digit | special_symbol | text_symbol .
digit = '0' | '1' | '2' | '3' | '4' |
        '5' | '6' | '7' | '8' | '9' .
letter = 'A-Z_and_a-z' .
space = 'space_bar' .
special_symbol = '_' | '"' | ''' | '(' | ')' | '.' | '<' | '>' |
                 '+' | '-' | '*' | '/' | '=' | '#' | ':' | ',' |
                 '&' | space | tab .
tab = 'tabulator' .
text_symbol = '!' | '$' | '%' | ';' | '?' | '@' | '[' | '\' |
              ']' | '^' | '`' | '{' | '|' | '}' | '~' .
*/

/* Simple definitions (italic synonyms) */

logical_expression = expression.
timing_simple_expression = simple_expression .

argument_list          = list .
attribute_list         = list .
index_list             = list .
object_identifier_list = list .

object_name_list       = name_list .
parameter_name_list    = name_list .
```

```
name_list =                    (     name { $ ',' name }     ) |
/*#+ unique list syntax */  ( '{' name { $ ',' name } '}' ) .

attribute_name         = name .
class_name             = name .
event_name             = name .
function_name          = name .
index_object_name      = name .
object_name            = name .
operating_system_name  = name .
parameter_name         = name .
procedure_name         = name .
time_zone_name         = name .
units_name =      /**- Added example unit names */
/**+*/ 'SECOND' | 'SECONDS' | 'CFM' | 'CM' | 'DEGC' | 'RPM' |
       'MINUTE' | 'MINUTES' | 'PERCENT' .
```

```
/*****************************************************************
 *
 *      UIL (User Interface Language) Extensions for SSLO2
 *                        Version 1.00
 *
 *****************************************************************/

UIL = { procedure | declaration | event_handler | extensions } .

/*  The action dictionary and object directory entries will be
 *  built and controlled by NASA.  For this example, they been
 *  coded into the syntax.  Note that defining the syntax for
 *  example_command in this way incorrectly allows such commands
 *  as OPEN MOTOR or LAUNCH VALVE.  In the operational system,
 *  the action dictionary will maintain a list of object classes
 *  that the action can be sent to.  */

example_command = example_action example_object .
example_action = action_dictionary_entry .
example_object = object_directory_entry .

action_dictionary_entry =
/* events/systems */      'enable' | 'disable'  | 'stop'  |
                'start' | 'revert' | 'enable_events_for'  |
/* operator */ 'write' | 'query'  |
/* systems */
    'chilldown_suction_line' | 'chilldown_transfer_line' |
    'chilldown_orbiter_mps'  | 'activate_topping'        |
    'open_main_fill_valve' .

object_directory_entry =
      'operator_object' | 'fill_system_object' |
      'active_pump'       | 'backup_pump'          |
      'stop_key_object' | 'revert_key_object' .

/*****************************************************************
 *
 *        UIL (User Interface Language) Object Description
 *                 Language Extensions for SSLO2
 *                        Version 1.00
 *
 *****************************************************************/

/*#+*/
extensions = ( 'DEFINE' ( big_object | big_class)  ) |
             send_action .

uil_action =
   'DEFINE' 'ACTION' name [ '(' parameter_name_list ')' ] 'IS'
      { $ sequence_of_statements }
   'END' 'DEFINE' 'ACTION' [ name ] .
```

```
big_class =
   'CLASS' name_list [ 'IS' name_list ]
      { $ declaration |
         ( 'ACTION'   ( name | action_dictionary_entry )
           'IS' ( ( name | action_dictionary_entry ) | 'NULL' ) ) }
      { $ uil_action }
   'END' 'DEFINE' 'CLASS' [ name_list ] .

big_object =
   'OBJECT' { name_list } 'IS' name_list
      { $ declaration |
         ( 'ACTION'   ( name | action_dictionary_entry )
           'IS' ( ( name | action_dictionary_entry ) | 'NULL' ) ) }
      { $ uil_action }
   'END' 'DEFINE' 'OBJECT' [ name_list ] .

/* Note that a more conventional syntax of object languages
   is allowed in this extended definition of the UIL syntax.
   Both forms shown below are functionally equivalent.  These
   more conventional forms allow any action and object to be used
   without conflicts or parsing problems.  Developers can use the
   formal forms for non-NASA controlled applications */

send_action = ( 'SEND' an_action 'TO' an_object) |
              ( 'ASK'  an_object 'TO' an_action) .
an_action = ( action_dictionary_entry | action ) [ with_clause ] .
an_object = 'SELF' | object_name .

$$$$$
```

APPENDIX B

GOAL GKH1F PROCEDURE LISTING

```
          BEGIN PROGRAM (GKH1F);

$         Identification
               System name -                      LH2
               S/W structure diagram -            80K00001
               Company / Group / Phone -          MMC/Software group
          Hardware/software configuration requirements
               Flight software -                  N/A
               Flight hardware -                  N/A
               PCM format -                       N/A
               LPS hardware                       LINK GSE 4
               GSE -                              PNEUMATIC VALVE A100677
                                                  SOLENOID VALVES A80995,
                                                  A80994,A80996.

          Function description
               Purpose/functions of program -
               The program is designed to command the A100677 main
               fill valve to the open position and set the appropriate
               exception monitor and GOAL notification limits.
               The following types of function designators are defined
               in the data base and are used in this program-

               <GLHK----E> Valve discrete commands
               <GLHX----E> Valve discrete position indicators
               <NLH------> Bypass function designators

               The respective bypasses are turned on whenever it is
               desired to inhibit the issuance of the command(s)
               to the component or if component response indicator(s)
               and/or signal(s) is determined erroneous. No bypasses
               are turned on within this program.

               OMRSD reqmts satisfied -         TBD
               Hazards and warnings -           TBD
               Prerequisites -                  TBD

          Comm FD's / function - SENT <N001LH2 $COM INTERRUPT$>
                                 TO CONSOLE <LH2 $RESP. SYSTEM CONSOLE$>
          Pseudo FD's / function -
               NLHK0101X- Fast flag - Is used when rapid execution is
                          required.
               NLHK9999X- Stop flow - Can be set on by the operator
               NLHK4111E- Stimulus bypass is tested to select primary
                          or secondary hardware operational mode or to
                          prevent issuance of a command.
               NLHX4112E, NLHX4113E, NLHX4123E -
                          An indicator/measurement bypass is tested for
                          data validity or for changing/activating in-
                          terrupt processing on a function designator.

               -additional comments deleted-.    $

          DECLARE QUANTITY (GMT1)=GMT, (GMT2)=GMT, (VLVTM)=SEC;
```

```
$         The previous items are internal variables for this subroutine
          only.      $

          SPECIFY INTERRUPT <PFPK6 $PSP KEY 6 DEFAULT$>
              AND ON OCCURRENCE GO TO STEP  40;

              $********** BEGIN OPERATING STEPS **********$
          ACTIVATE PROCEDURE ERROR OVERRIDE;

          CHANGE <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$>
               <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$>
               <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
               <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>
               <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
               SAMPLE RATE TO 100 TIMES PER SECOND;

          INHIBIT EXCEPTION MONITORING FOR
               <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
               <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
               <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>;

          INHIBIT FEP INTERRUPT CHECK FOR
               <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
               <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
               <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>;

$         The previous statements inhibit those interrupts which
          would result from this valve changing state.  All other
          interrupts are activated.  $

$         The following statements check each valve position indicator
          bypass.  If the bypass is on, no action is taken.  If the
          bypass is off, GOAL and system exception conditions are
          changed to reflect the state each indicator is expected to
          show upon program completion.          $

          VERIFY <NLHX4112E $A100677 MAIN FILL VALVE CL IND BYP$> IS OFF
               ELSE GO TO STEP   2;

          CHANGE <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
               GOAL EXCEPTION CONDITION TO ON;

          CHANGE <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
               SYSTEM EXCEPTION CONDITION TO ON;

STEP    2 VERIFY <NLHX4123E $A100677 MAIN FILL VLV RED IND BYP$> IS OFF
               ELSE GO TO STEP   3;

          CHANGE <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
               GOAL EXCEPTION CONDITION TO ON;

          CHANGE <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
               SYSTEM EXCEPTION CONDITION TO ON;
```

```
STEP   3 VERIFY <NLHK0101X $FAST FLAG$> IS OFF
              ELSE GO TO STEP    4;

         VERIFY <NLHX4113E $A100677 MAIN FILL VALVE OP IND BYP$> IS OFF
              ELSE GO TO STEP 100;

         CHANGE <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>
              GOAL EXCEPTION CONDITION TO OFF;

         CHANGE <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>
              SYSTEM EXCEPTION CONDITION TO OFF;

STEP 100 ACTIVATE INTERRUPT PROCESSING ON THIS LEVEL;

$        The valve is commanded to its open position and the program
         verifies that prerequisite control logic allowed the commands
         to be issued.          $

STEP   4 VERIFY <NLHX4112E $A100677 MAIN FILL VALVE CL IND BYP$> IS OFF
              THEN GO TO STEP    5;

         RECORD <GMT $GREENWICH MEAN TIME$>
              FORMAT (NO UNITS,NO FD NAME,NO FD DESCRIPTOR)
              , TEXT ( GKH1F- VALVE A100677 OPEN CMD GLHK4111E)
              NEXT TEXT (       IS BYPASSED, PROGRAM TERMINATED)
              TO <PAGE-A $DISPLAY APPLICATION PAGE B$> YELLOW
              TO <CNSL-PP $CONSOLE PRINTER PLOTTER$>
              <SPA-PRNTR $SPA PRINTER$>;

         GO TO STEP 15;

STEP   5 TURN ON <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$>;

         TURN OFF <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$>;

         VERIFY <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$> IS ON
           AND <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$> IS OFF
              ELSE GO TO STEP   15;

$        The following timing loop allows 6 seconds in which to
         establish initial component motion and up to 20 seconds
         for the valve to home.  When NLHK0101X fast flag is on
         the program terminated after initial component motion has
         been established.          $

         READ <GMT $GREENWICH MEAN TIME$> AND SAVE AS (GMT1);

STEP   6 READ <GMT $GREENWICH MEAN TIME$> AND SAVE AS (GMT2);

         LET (VLVTM) = (GMT2) - (GMT1);

         IF (VLVTM) IS LESS THAN 0.0 SEC,
              LET (VLVTM) = (VLVTM) + 86400 SEC;
```

```
             IF (VLVTM) IS GREATER THAN OR EQUAL TO 6 SEC,
                  THEN GO TO STEP    9;

             VERIFY <NLHX4112E $A100677 MAIN FILL VALVE CL IND BYP$> IS OFF
                  ELSE GO TO STEP    7;

             VERIFY <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$> IS OFF
                  ELSE GO TO STEP    6;

             GO TO STEP 10;

STEP     7   VERIFY <NLHX4123E $A100677 MAIN FILL VLV RED IND BYP$> IS OFF
                  ELSE GO TO STEP    8;

             VERIFY <GLHX4123E $A100677 MAIN FILL VALVE RED IND$> IS OFF
                  ELSE GO TO STEP    6;

             GO TO STEP 10;

STEP     8   RECORD <GMT $GREENWICH MEAN TIME$>
                  FORMAT (NO UNITS,NO FD NAME,NO FD DESCRIPTOR)
                  , TEXT ( GKH1F- VLV A100677 CLOSE IND GLHX4112E AND RDCD)
                  NEXT TEXT (       IND GLHX4123E ARE BYP)
                  NEXT TEXT (       INITIAL MOTION CANNOT BE DETERMINED)
                  TO <PAGE-A $DISPLAY APPLICATION PAGE B$> YELLOW
                  TO <CNSL-PP $CONSOLE PRINTER PLOTTER$>
                  <SPA-PRNTR $SPA PRINTER$>;

             GO TO STEP 10;

STEP     9   RECORD <GMT $GREENWICH MEAN TIME$>
                  FORMAT (NO UNITS,NO FD NAME,NO FD DESCRIPTOR)
                  , TEXT ( GKH1F- VLV A100677 INITIAL MOTION IS GREATER ),
                  NEXT TEXT (       THAN 6 SEC)
                  TO <PAGE-A $DISPLAY APPLICATION PAGE B$> YELLOW
                  TO <CNSL-PP $CONSOLE PRINTER PLOTTER$>
                  <SPA-PRNTR $SPA PRINTER$>;

             GO TO STEP 10;

STEP    10   VERIFY <NLHK0101X $FAST FLAG$> IS OFF
                  ELSE GO TO STEP   15;

STEP    11   READ <GMT $GREENWICH MEAN TIME$> AND SAVE AS (GMT2);

             LET (VLVTM) = (GMT2) - (GMT1);

             IF (VLVTM) IS LESS THAN 0.0 SEC,
                  LET (VLVTM) = (VLVTM) + 86400 SEC;

             IF (VLVTM) IS LESS THAN 20 SEC,
                  THEN GO TO STEP   13;
```

```
         VERIFY <NLHX4113E $A100677 MAIN FILL VALVE OP IND BYP$> IS OFF
              ELSE GO TO STEP   12;

         RECORD <GMT $GREENWICH MEAN TIME$>
              FORMAT (NO UNITS,NO FD NAME,NO FD DESCRIPTOR)
              , TEXT ( GKH1F- VALVE A100677 OPEN TIME EXCEEDED LIMITS),
              TO <PAGE-A $DISPLAY APPLICATION PAGE B$> YELLOW
              TO <CNSL-PP $CONSOLE PRINTER PLOTTER$>
              <SPA-PRNTR $SPA PRINTER$>;

         GO TO STEP 15;

STEP  12 RECORD <GMT $GREENWICH MEAN TIME$>
              FORMAT (NO UNITS,NO FD NAME,NO FD DESCRIPTOR)
              , TEXT ( GKH1F- VLV A100677 OPEN IND GLHX4113E BYPASSED),
              TO <PAGE-A $DISPLAY APPLICATION PAGE B$> YELLOW
              TO <CNSL-PP $CONSOLE PRINTER PLOTTER$>
              <SPA-PRNTR $SPA PRINTER$>;

         GO TO STEP 15;

STEP  13 VERIFY <NLHX4113E $A100677 MAIN FILL VALVE OP IND BYP$> IS OFF
              ELSE GO TO STEP   11;

         VERIFY <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$> IS ON
              ELSE GO TO STEP   11;

         RECORD <GMT $GREENWICH MEAN TIME$>
              FORMAT (NO UNITS,NO FD NAME,NO FD DESCRIPTOR)
              , TEXT ( GKH1F- VALVE A100677 OPEN TIME IS),
              (VLVTM)
              TO <PAGE-A $DISPLAY APPLICATION PAGE B$> YELLOW
              TO <CNSL-PP $CONSOLE PRINTER PLOTTER$>
              <SPA-PRNTR $SPA PRINTER$>;

 $       Interrupts are activated for those valve indicators which are
         not bypassed.       $

         VERIFY <NLHX4112E $A100677 MAIN FILL VALVE CL IND BYP$> IS OFF
              ELSE GO TO STEP   17;

         ACTIVATE EXCEPTION MONITORING
              FOR <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>;

         ACTIVATE FEP INTERRUPT CHECK
              FOR <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>;

STEP  17 VERIFY <NLHX4123E $A100677 MAIN FILL VLV RED IND BYP$> IS OFF
              ELSE GO TO STEP   18;

         ACTIVATE EXCEPTION MONITORING
              FOR <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>;
```

```
          ACTIVATE FEP INTERRUPT CHECK
               FOR <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>;

STEP  18  VERIFY <NLHX4113E $A100677 MAIN FILL VALVE OP IND BYP$> IS OFF
               ELSE GO TO STEP   19;

          ACTIVATE EXCEPTION MONITORING
               FOR <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>;

          ACTIVATE FEP INTERRUPT CHECK
               FOR <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>;

STEP  19  CHANGE <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$>
               <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$>
               <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
               <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>
               <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
               SAMPLE RATE TO 0 TIMES PER SECOND;

          TERMINATE;

STEP  40  TURN ON <NLHK9999X $STOP FLOW$>;

          CHANGE <GLHK4111ER $HRS A100677 MAIN FILL VLV OPEN CMD$>
               <GLHK4121ER $HRS A100677 MAIN FILL VLV REDU CMD$>
               <GLHX4112E $A100677 MAIN FILL VALVE CLOSED IND$>
               <GLHX4113E $A100677 MAIN FILL VALVE OPEN IND$>
               <GLHX4123E $A100677 MAIN FILL VALVE RED IND$>
               SAMPLE RATE TO 0 TIMES PER SECOND;

          SEND INTERRUPT <N001LH2 $COM INTERRUPT$>
               TO CONSOLE <LH2 $RESP. SYSTEM CONSOLE$>;

          TERMINATE;

          END PROGRAM;
```

APPENDIX C

UIL SSLO2 HIGH LEVEL CONTROL PROGRAM

```
UIL
/****************************************************************
 ****************************************************************
 *                  SSLO2 routine written in UIL
 *                  This version translated by G. T. Sos
 *                  from KSC GOAL SSLO2 code
 *
 * SSLO2 LOX Auto Fill Sequencer:
 *  This procedure performs the operations necessary to fill the LOX
 *  (liquid oxygen) portion of the ET (External tank).  It remains
 *  in execution until the 100% liquid level sensors flash wet
 *  (show tank is 100% full).
 *
 *  First, setup events and perform an instrumentation status
 *  check.  Next, perform the chilldown of the LOX pump suction
 *  line, pump transfer line, orbiter MPS (Main Propulsion
 *  System).  Once chilldown is complete, perform slow fill
 *  external LOX tank to 2%, fast fill external LOX tank to
 *  98% and topping external LOX tank 100%.  SSLO2 then
 *  transitions to replenish program to keep the external tank
 *  topped off to between 100% and 100.15%.
 *
 ****************************************************************
 ****************************************************************/

/* NOTE: Upper case are language syntax keywords */

PROCEDURE sslo2_procedure IS
   CONSTANT  long_name = "Program unit SSLO2"
   PARAMETER active_pump    IS hwsubsystem_class
   PARAMETER backup_pump    IS hwsubsystem_class
   PARAMETER temp           IS basic_object_class
   PARAMETER sslo2_status   IS ENUMERATED
                            WITH VALUES = {'run', 'stop', 'revert'}

/****************************************************************
 *
 *                        Specify exceptions
 *
 ****************************************************************/

disable stop_key_object
disable revert_key_object

EVENT HANDLER FOR stop_key_object IS
      enable   revert_key_object
      disable  stop_key_object
      stop     fill_system_object
      LET      sslo2_status = 'stop'
      query    operator_object WITH options = {'OK', 'quit'}
   END stop_key_object
```

```
EVENT HANDLER FOR revert_key_object IS
      disable  revert_key_object
      enable   stop_key_object
      revert   fill_system_object
      LET      sslo2_status = 'revert'
      query    operator_object WITH options = {'OK', 'quit'}
   END revert_key_object

LET sslo2_status = 'run'
enable stop_key_object

enable_events_for fill_system_object

/*****************************************************************
 *
 *                      Begin operating steps
 *
 *  The original SSLO2 has additional code for stopping and
 *  restarting the filling of the LO2 external tank that was not
 *  included in the UIL version.
 *
 *****************************************************************/

write operator_object WITH output =
   "SSLO2 procedure execution begun"

/* Select a pump */

query operator_object WITH output  = "Select main p128 pump or
                                      alternate a128 pump",
                           options = {'main', 'alternate', 'quit'}

IF response OF operator_object = 'quit' THEN RETURN END IF

IF response OF operator_object = 'main'
    THEN
       LET active_pump = main_pump      OF fill_system_object
       LET backup_pump = alternate_pump OF fill_system_object
    OTHERWISE
       LET active_pump = alternate_pump OF fill_system_object
       LET backup_pump = main_pump      OF fill_system_object
END IF

enable  active_pump
disable backup_pump

/* Setup and instrumentation status check (120 GOAL statements) */
/* Actions are be expected to report any anomalies to operator */

check_status fill_system_object WITH output_to = operator_object
```

```
query operator_object WITH output  =
   "Instrument check complete, holding for pump start",
                             options = {'OK'}

SET speed OF active_pump TO 1000 RPM
start active_pump
WAIT startup_time OF active_pump
     OR UNTIL command_completed OF active_pump

query operator_object WITH output  =
   "Pump started, holding for chilldown",
                             options = {'OK'}

/* LOX pump suction line chilldown (156 GOAL statements) */

chilldown_suction_line active_pump
WAIT suction_line_cool OF active_pump

/* Pump transfer line and vehicle chilldown
   (240 GOAL statements) */

chilldown_transfer_line active_pump
WAIT transfer_line_cool OF active_pump

/* Orbiter MPS (Main Propulsion System) chilldown
   (438 GOAL statements) */

chilldown_orbiter_mps fill_system_object
WAIT orbiter_mps_cool OF fill_system_object

/* Slow fill external LOX tank to 2% (84 GOAL statements) */

query operator_object WITH output  =
   "Chilldown done, holding for  pump speed ramp to 2850 RPM",
                             options = {'OK'}

SET speed OF active_pump TO 2850 RPM
WAIT UNTIL command_completed OF active_pump

query operator_object WITH output  =
   "Pump ramped to 2850 RPM, holding for 2% fill",
                             options = {'OK'}

SET flash_point_level OF fill_system_object TO 2 PERCENT
open_main_fill_valve active_pump

WAIT slow_fill_time_limit OF fill_system_object
   OR UNTIL flash_point OF et_tank_status_object

query operator_object WITH output  =
   "2% slow fill complete, holding for 98% fast fill",
                             options = {'OK'}
```

```
/* Fast fill external LOX tank 2% to 98% (205 GOAL statements) */

SET flash_point_level OF fill_system_object TO 98 PERCENT
SET speed OF active_pump TO 3450 RPM

WAIT UNTIL command_completed OF active_pump

WAIT fast_fill_time_limit OF fill_system_object
   OR UNTIL flash_point OF et_tank_status_object

query operator_object WITH output =
   "2% slow fill complete, holding for 98% fast fill",
                              options = {'OK'}

write operator_object WITH output =
   "98% fill complete, ramping pump to 2850 RPM for topping"

/* Topping external LOX tank 98% to 100% (228 GOAL statements) */

SET flash_point_level OF fill_system_object TO 100 PERCENT
SET speed OF active_pump TO 2850 RPM
WAIT UNTIL command_completed OF active_pump

WAIT complete_fill_time_limit OF fill_system_object
   OR UNTIL flash_point OF et_tank_status_object

/* GOAL: Transfer to replenish program (4 GOAL statements) */

write operator_object WITH output =
   "100% fill complete, transferring to topping"

activate_topping fill_system_object

END

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

$$$$$
```

APPENDIX D

UIL SSLO2 OBJECT/ACTION DEFINITIONS

```
UIL
/*****************************************************************
 *****************************************************************
 *                    System Object Class Definitions
 *
 *   Object classes defined in detail: basic_object_class,
 *   pneumatic_valve_class, pneumatic_776_valve_class AND
 *   liquid_oxygen_precautions_class.
 *
 *****************************************************************
 *****************************************************************/


/*****************************************************************
 *                 Define the Basic Object Actions
 *
 *  All objects used for SSLO2 have basic actions that will be
 *  provided by default.  These object action definitions are
 *  shown below.
 *
 *****************************************************************/

DEFINE CLASS basic_object_class

/*  The actions for the basic object will not be described in
    detail.  They perform at least the following functions:

 new         - Called when the object is created.
 null        - An action that does nothing.
 print       - Prints information about the object to a I/O stream.
 describe    - Types information about the object to the operator.
 list_actions - Returns list of the actions that object can handle.
 is_action   - Returns true if this is a actions of this object.
 unhandled_action - Handles action requests not defined by the object.
 destroy     - Called when object is destroyed. */

END DEFINE CLASS basic_object_class

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

$$$$$
```

```
UIL
/*****************************************************************
 *
 * Very simple valve object
 * with just 3 bits of hardware information!
 *  open_status, closed_status and command_status
 *
 *    Read hardware bit    Bit set                Bit clear
 *  open_status        valve is open            valve not open
 *  closed_status      valve is closed          valve not closed
 *  command_status     commanded to open        commanded to close
 *
 *    Write hardware bit    Bit set               Bit clear
 *  command_status     command to open          command to close
 *
 *    Software bit                Bit set               Bit clear
 *  open_status_bypassed     Ignore open status      Use status
 *  close_status_bypassed    Ignore closed status    Use status
 *  command_status_bypassed  Ignore command status   Use status
 *
 *  The valve is designed such that it contains two switches:
 *  one can detect when the valve has been completely opened,
 *  and the other shows when it is completely closed.  When the
 *  valve is in transition from open to close, then neither
 *  switch is activated.  These two status switches and the
 *  bypass flags make the valve quite complicated to control.
 *
 *****************************************************************/

DEFINE CLASS pneumatic_valve_class IS basic_object_class
   CONSTANT long_name = "pneumatic valve"
   CONSTANT type = "basic class"
   CONSTANT initial_component_motion = 1 SECOND .. 2 SECONDS
   CONSTANT home_time = 5 SECONDS .. 6 SECONDS

   ACTION new          IS new_procedure
   ACTION initial      IS initial_procedure
   ACTION open         IS open_procedure
   ACTION close        IS close_procedure
   ACTION stop         IS stop_procedure
   ACTION revert       IS revert_procedure
   ACTION enable       IS enable_procedure
   ACTION disable      IS disable_procedure
   ACTION warning      IS warning_procedure
   ACTION dynamics     IS dynamics_procedure
   ACTION destroy      IS destroy_procedure
   PARAMETER {input, output} IS basic_object_class
   PARAMETER {last_state, desired_state, current_state}
         IS ENUMERATED WITH VALUES = {'open', 'closed', 'unknown'}
   PARAMETER is_opened IS ENUMERATED WITH VALUES =
                              {'opened', 'not opened'}
   PARAMETER is_closed IS ENUMERATED WITH VALUES =
                              {'closed', 'not closed'}
   PARAMETER is_stopped IS ENUMERATED WITH VALUES =
                              {'stopped', 'not stopped'}
   PARAMETER transition_status IS ENUMERATED WITH VALUES =
       {'waiting for component motion', 'completed',
        'waiting for component home'}
   PARAMETER {open_status_bypassed, close_status_bypassed,
       command_status_bypassed}          IS shared_memory_class
   PARAMETER {open_status, closed_status} IS readable_hw_port_class
   PARAMETER command_status              IS readwriteable_hw_port_class

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

DEFINE ACTION new_procedure IS
   /* executed only at instantiation or object */
   LET open_status_bypassed    = 'off'
   LET close_status_bypassed    = 'off'
   LET command_status_bypassed = 'off'
   ASK SELF TO warning_procedure WITH new_level='disable all'
   LET last_state    = 'unknown'
   LET desired_state = 'unknown'
   LET current_state = 'unknown'
   LET transition_status = 'completed'
   LET is_opened     = 'unknown'
   LET is_closed     = 'unknown'
   LET is_stopped    = 'not stopped'
END DEFINE ACTION new_procedure

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

DEFINE ACTION initial_procedure IS
   PARAMETER temp_current_state IS ENUMERATED
          WITH VALUES = { 'opened', 'likely opened', 'closed',
                'likely closed', 'bypassed', 'maybe open',
                'maybe closed' , 'conflicting', 'in transition' }

   IF {open_status_bypassed, close_status_bypassed,
          command_status_bypassed} /= 'off'
      THEN
           query operator_object WITH output =
              "Warning - valve open or close bypass on engaged!",
                      options = {'OK'}
      END IF

   LET temp_hardware_state = get_hardware_state_procedure

   CASE temp_hardware_state
```

```
     WHEN  'opened' OR 'likely opened' OR 'closed'
           OR 'likely closed'
        THEN NULL

     WHEN  'maybe open' OR 'maybe closed'
        THEN
           query operator_object WITH output =
              "Warning - low certainty of valve position!",
                                  options = {'OK'}

     WHEN 'unknown' OR 'conflicting' OR 'in transition'
        THEN
           query operator_object WITH output =
              "Error - valve position undetermined!",
                                  options = {'OK'}

   END CASE

   ASK SELF TO warning_procedure WITH new_level='disable all'
   LET last_state = 'unknown'
   LET desired_state = temp_current_state
   LET current_state = temp_current_state
   LET transition_status = 'completed'
   LET is_opened = temp_is_opened
   LET is_closed = temp_is_closed
   LET is_stopped = 'not stopped'
   ASK SELF TO warning_procedure WITH new_level =
                               'maintain current state'
END DEFINE ACTION initial_procedure

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

DEFINE ACTION open_procedure ( fast_mode ) IS
   PARAMETER fast_mode IS ENUMERATED WITH VALUES =
                            {'stopped', 'not stopped'}
   PARAMETER temp_hardware_state IS ENUMERATED
         WITH VALUES = { 'opened', 'likely opened', 'closed',
                'likely closed', 'bypassed', 'maybe open',
                'maybe closed' , 'conflicting', 'in transition' }

   LET temp_hardware_state =
      report_hardware_state_procedure

   IF {open_status_bypassed, close_status_bypassed, fast_mode,
      command_status_bypassed, temp_hardware_state} =
      {'off', 'off', 'off', 'off', 'closed'} THEN

         /* No bypassed status bits, do complete open */
         IF {current_state, temp_hardware_state} = 'open'
            THEN
               query operator_object WITH output =
                  "open request on open valve",
                     options = {'OK'}
            END IF

      LET desired_state = 'open'
      ASK SELF TO warning_procedure WITH new_level='allow to open'

      LET transition_status = 'waiting for component motion'
      LET command_status = 'open'

      WAIT UPPER(initial_component_motion)
           OR UNTIL close_status = 'not closed'

      IF close_status /= 'not closed' THEN
          query operator_object WITH output  = "Error",
                                     options = {'OK'}
      END IF

      LET transition_status = 'waiting for component home'

      WAIT UPPER(home_time)
           OR UNTIL close_status = 'open'

      IF close_status /= 'open' THEN
          query operator_object WITH output  = "Error",
                                     options = {'OK'}
      END IF

      ASK SELF TO warning_procedure WITH new_level='maintain open'

      OTHERWISE
          /* ADD bypassed status open procedure here */
          NULL
      END IF

   LET last_state = current_state
   LET current_state = 'open'
   LET transition_status = 'completed'
   RETURN 'ok'
END DEFINE ACTION open_procedure

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

DEFINE ACTION close_procedure IS
   /* similar to open */
END DEFINE ACTION close_procedure

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
DEFINE ACTION stop_procedure IS
   /* handles case of stop operation from operator */
END DEFINE ACTION stop_procedure

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

DEFINE ACTION revert_procedure IS
   /* handles case of revert (restart) operation from stop */
END DEFINE ACTION revert_procedure

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

DEFINE ACTION enable_procedure IS
   /* handles enable message.  In this example, enables this
      valve for any operations. */
END DEFINE ACTION enable_procedure

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

DEFINE ACTION disable_procedure IS
   /* handles disable message.  In this example, disables
      this valve for any operations. */
END DEFINE ACTION disable_procedure

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

DEFINE ACTION get_hardware_state_procedure IS
   PARAMETER {temp_hardware_state, temp_is_opened, temp_control_status,
              temp_is_closed, temp_control_command} IS ENUMERATED
         WITH VALUES = { 'opened', 'likely opened', 'closed',
                'likely closed', 'bypassed', 'maybe open',
                'maybe closed' , 'conflicting', 'in transition' }

   /* Attempt to determine current state from hardware without
      interrupting current operations. read values of sensors
      into temporaries and analyze the results */

   LET temp_hardware_state = 'unknown'
   LET temp_is_opened = 'bypassed'
   LET temp_is_closed = 'bypassed'
   LET temp_control_status = 'bypassed'
   IF open_status_bypassed = 'off' THEN
      LET temp_is_opened = open_status
      END IF
   IF close_status_bypassed = 'off' THEN
      LET temp_is_closed = close_status
      END IF
   IF command_status_bypassed = 'off' THEN
      LET temp_control_status = command_status
      END IF
```

```
/* the following is an example of how to handle partially known
   state of the hardware.  This code is NOT in the actual SSLO2
   program, as it is my own ideas of how this might be handled with
   an enumerated type.  SSLO2 handled the problem on a case by case
   basis and, in general, only checked a few of the possible
   states.  */

 CASE {temp_is_opened, temp_is_closed, temp_control_status}

  WHEN  {'opened',         'not closed',       'commanded open'}
   THEN LET temp_hardware_state='opened' /* 3 votes in 3 */

  WHEN  {'opened',         'bypassed',         'commanded open'} OR
        {'bypassed',       'not closed',       'commanded open'} OR
        {'opened',         'not closed',       'bypassed'}
   THEN LET temp_hardware_state='likely opened' /* 2 votes in 2 */

  WHEN  {'bypassed',       'bypassed',         'commanded open'} OR
        {'opened',         'bypassed',         'bypassed'} OR
        {'bypassed',       'not closed',       'bypassed'}
   THEN LET temp_hardware_state='maybe open' /* 1 vote in 1 */

  WHEN  {'not opened',     'closed',           'commanded closed'}
   THEN LET temp_hardware_state='closed' /* 3 votes in 3 */

  WHEN  {'not opened',     'bypassed',         'commanded closed'} OR
        {'bypassed',       'closed',           'commanded closed'} OR
        {'not opened',     'closed',            'bypassed'}
   THEN LET temp_hardware_state='likely closed' /* 2 votes in 2 */

  WHEN  {'bypassed',       'bypassed',         'commanded closed'} OR
        {'bypassed',       'closed',           'bypassed'} OR
        {'not opened',     'bypassed',         'bypassed'}
   THEN LET temp_hardware_state='maybe closed' /* 1 vote in 1 */

  WHEN  {'opened',         'not closed',       'commanded closed'} OR
        {'not opened',     'not closed',       'commanded closed'} OR
        {'opened',         'bypassed',         'commanded closed'} OR
        {'bypassed',       'not closed',       'commanded closed'} OR
        {'not opened',     'bypassed',         'commanded open'} OR
        {'bypassed',       'closed',           'commanded open'} OR
        {'not opened',     'not closed',       'commanded open'} OR
        {'not opened',     'closed',           'commanded open'} OR
        {'not opened',      'not closed',       'bypassed'}
   THEN LET temp_hardware_state='in transition' /* 2 votes in 3 */

  WHEN  {'opened',         'closed',           'commanded closed'} OR
        {'opened',         'closed',           'commanded open'} OR
        {'opened',         'closed',           'bypassed'}
   THEN LET temp_hardware_state='conflicting' /* 1 vote in 2 */
  WHEN  {'bypassed',      'bypassed',        'bypassed'}
   THEN LET temp_hardware_state='bypassed' /* 0 votes in 0 */
 END CASE

RETURN temp_hardware_state
END DEFINE ACTION get_hardware_state_procedure

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

DEFINE ACTION report_hardware_state_procedure IS
   PARAMETER temp_hardware_state IS ENUMERATED
         WITH VALUES = { 'opened', 'likely opened', 'closed',
                'likely closed', 'bypassed', 'maybe open',
                'maybe closed' , 'conflicting', 'in transition' }

   LET temp_hardware_state = get_hardware_state_procedure

   CASE temp_hardware_state

     WHEN 'opened' OR 'likely opened' OR
          'closed' OR 'likely closed'
        THEN NULL

     WHEN 'bypassed' OR 'maybe open' OR 'maybe closed'
        THEN
           query operator_object WITH output  =
              "Warning low certainty of valve position!",
                                  options = {'OK'}

    WHEN 'conflicting' OR 'in transition'
        THEN
           query operator_object WITH output  =
              "Error valve position undetermined!",
                                  options = {'OK'}
   END CASE
RETURN temp_hardware_state
END DEFINE ACTION report_hardware_state_procedure

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

DEFINE ACTION warning_procedure(new_level) IS
   PARAMETER new_level IS ENUMERATED WITH VALUES =
           {'maintain current state', 'allow transition to open',
            'allow transition to closed', 'maintain open',
            'maintain close', 'disable all'}
   PARAMETER status IS ENUMERATED WITH VALUES =
        {'waiting for component motion', 'completed',
         'waiting for component home'}
   IF new_level = 'maintain current state' THEN
      IF current_state = 'open' THEN
         LET new_level = 'maintain open'
         OTHERWISE LET new_level = 'maintain close'
      END IF
   END IF

   CASE new_level
      WHEN  'allow transition to open' OR
            'allow transition to close' THEN
         EVENT HANDLER FOR open_status    IS NULL END
         EVENT HANDLER FOR close_status   IS NULL END
         EVENT HANDLER FOR command_status IS NULL END
      WHEN  'maintain open' OR 'maintain close' THEN
         IF open_status_bypassed = 'off' THEN
            EVENT HANDLER FOR open_status IS
               LET status = state_change_handler() END
         END IF
         IF close_status_bypassed = 'off' THEN
            EVENT HANDLER FOR closed_status IS
               LET status = state_change_handler() END
         END IF
         IF command_status_bypassed = 'off' THEN
            EVENT HANDLER FOR command_status IS
               LET status = state_change_handler() END
         END IF
      WHEN  'disable all' THEN
         EVENT HANDLER FOR open_status    IS NULL END
         EVENT HANDLER FOR close_status   IS NULL END
         EVENT HANDLER FOR command_status IS NULL END
   END CASE
END DEFINE ACTION warning_procedure

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

DEFINE ACTION state_change_handler ( event_source ) IS
   IF event_source THEN
        query operator_object WITH output  =
             "Error condition unexpected change of state!",
                                       options = {'OK'}
   END IF
END DEFINE ACTION state_change_handler

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

DEFINE ACTION dynamics_procedure IS
   /* the dynamics procedure would contain all the necessary
      information for simulations of the valve, and of the
      valves failure modes (not included here) */
END DEFINE ACTION dynamics_procedure

END DEFINE CLASS pneumatic_valve_class
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
$$$$$
```

```
UIL
/****************************************************************
 *               Define New Valve Based on Existing Class       *
 *                                                              *
 ****************************************************************/

DEFINE CLASS pneumatic_a776_valve_class IS pneumatic_valve_class
   REPLACE  long_name = "model A776 pneumatic valve"
   REPLACE  initial_component_motion = 2 SECONDS .. 6 SECONDS
   REPLACE  home_time = 18 SECONDS .. 20 SECONDS
   REPLACE  maximum_flow_rate = 35 CFM
   REPLACE  diameter = 45 CM

/* all other actions and object data inherited from
  pneumatic_valve_class class */

END DEFINE CLASS pneumatic_valve_class

/****************************************************************
 *               Simple Liquid Oxygen Precautions Object
 *
 *  This object is intended to provide liquid oxygen specific
 *  information for other objects.  This object provides a good
 *  way to consistently apply a set of related information to
 *  other classes.
 *
 ****************************************************************/

DEFINE CLASS liquid_oxygen_precautions_class IS basic_object_class
   CONSTANT long_name = "oxygen precautions and alerts"
   CONSTANT type = "precautions class"
   CONSTANT boiling_point = -182.962 DEGC
   CONSTANT safe_chilldown_temp = -186.0 DEGC

   ACTION new        IS new_procedure
   ACTION warning    IS warning_procedure
   ACTION dynamics   IS dynamics_procedure

   PARAMETER
      {last_state, desired_state, current_state}
         IS ENUMERATED WITH VALUES =
                  {'maintain current state',
                   'allow transition to warm',
                   'allow transition to chilldown',
                   'maintain warm',
                   'maintain chilldown',
                   'disable all'}

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

DEFINE ACTION new_procedure IS
   LET current_state = 'disable all'
END DEFINE ACTION new_procedure
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

DEFINE ACTION warning_procedure(new_level) IS
   PARAMETER new_level IS ENUMERATED WITH VALUES =
          { 'maintain current state',
            'allow transition to warm',
            'allow transition to chilldown',
            'maintain warm', 'maintain chilldown', 'disable all'}

/* This code would manage the warning messages and state changes for
   the variables contained in this object class */
END DEFINE ACTION warning_procedure

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

DEFINE ACTION state_change_handler ( event_source ) IS
   IF event_source THEN
        query operator_object WITH output  =
           "Error condition unexpected change of state!",
                                   options = {'OK'}
   END IF
END DEFINE ACTION state_change_handler

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

DEFINE ACTION dynamics_procedure IS
 /* the dynamics procedure would add all the necessary
    information for simulations of the objects (like pipes,
    pumps, ...), and of any failure modes */
END DEFINE ACTION dynamics_procedure

END DEFINE CLASS liquid_oxygen_precautions_class

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

$$$$$
```

```
UIL
/****************************************************************
 ****************************************************************
 *                    System Object Instantiations
 *
 *  These object instantiations would, most likely, be build and
 *  stored as controlled objects by NASA.  It is also presumed that
 *  tools will exist that will allow graphical and textual
 *  editing of these objects; therefore, it is unlikely that the
 *  SSLO2 subsystem would be created by the object list
 *  like the one that follows.
 *
 *  Four different examples of object classes are
 *  defined in detail: pneumatic_valve_class,
 *  pneumatic_776_valve_class, basic_object_class AND
 *  liquid_oxygen_precautions_class.
 *
 *  Other object CLASSES used but not defined in detail include:
 *    hwsubsystem_class       IS "hwsubsystem with operator I/O"
 *    plus100_hp_motor_class  IS "100 horse power pump"
 *    plus100_hp_clutch_class IS "100 horse power clutch"
 *    plus10_cm_pump_class    IS "smaller H2O pump"
 *    plus45_cm_pump_class    IS "large LOX pump"
 *    temp_sensor_class       IS "standard temperature sensor"
 *    tachometer_class        IS "standard tachometer"
 *    control_point_class     IS "hardware control point"
 *    high_temp_sensor_class  IS "hw sensor A"
 *    low_temp_sensor_class   IS "hw sensor B"
 *    low_tachometer_class    IS "hw sensor C"
 *
 *  Other hardware OBJECTS used but not defined in detail include:
 *   l1_object         IS "700K liters LOX tank sensor object"
 *   byp_l1_object     IS "operator bypass for l1_object object"
 *   t9_object         IS "-200 to -170 degc temp sensor object"
 *   byp_t9_object     IS "operator bypass for t9_object object"
 *   sslo2_lox_tank_object IS "700K liters LOX tank object"
 *
 *  Other OBJECTS used but not defined in detail include:
 *   revert_key_object      IS "operator software key for revert
 *   stop_key_object        IS "operator software key for stop"
 *   operator_object        IS "operator terminal output/input"
 *
 ****************************************************************
 ****************************************************************/

DEFINE OBJECT sslo2_pump_hwsubsystem_object IS hwsubsystem_class
   REPLACE  long_name = "Primary pump128 hwsubsystem"
   CONSTANT contains  = {t1_object, t2_object, t3_object,
                         t4_object, t5_object, t6_object,
                         t7_object, t8_object, r1_object,
                         motor_p128_object, clutch_p128_object,
                         h2o_pump_p128_object, pump_p128_object,
                         valve_p128_object}
END DEFINE OBJECT sslo2_pump_hwsubsystem_object
```

```
DEFINE OBJECT valve_p128_object IS
   {pneumatic_a776_valve_class, sslo2_pump_hwsubsystem_class,
    liquid_oxygen_precautions_class}
   REPLACE  long_name          = "Main LOX fill valve"
   CONSTANT manufacture_type = "Rockwell 30544766E5G"
   CONSTANT serial_number     = "878-32487-34985837-48 89.02.04"
   CONSTANT input             = pump_p128_object
   CONSTANT output            = orbiter_et_main_object
   PARAMETER {open_status_bypassed, close_status_bypassed,
              command_status_bypassed} IS shared_memory_class
        /* set the valves of objects to actual
           shared memory locations here */
   PARAMETER {open_status, closed_status, command_status}
                                       IS readable_hw_port_class
        /* set the valves of objects to actual hardware
           ports here */
END DEFINE OBJECT valve_p128_object

DEFINE OBJECT      {byp_t1_object, byp_t2_object, byp_t3_object,
                    byp_t4_object, byp_t5_object, byp_t6_object,
                    byp_t7_object, byp_t8_object, byp_r1_object}
                        IS shared_memory_class
END DEFINE OBJECT {byp_t1_object, byp_t2_object, byp_t3_object,
                    byp_t4_object, byp_t5_object, byp_t6_object,
                    byp_t7_object, byp_t8_object, byp_r1_object}

DEFINE OBJECT t1_object IS high_temp_sensor_class
   REPLACE  long_name   = "0 TO 200 DEGC Temp sensor"
   CONSTANT bypass_flag = byp_t1_object
      /* set the valves of object variables to actual
         hardware ports here */
END DEFINE OBJECT t1_object
/* objects t2_object thru t6_object defined similar to t1_object */

DEFINE OBJECT t7_object IS
 low_temp_sensor_class
   REPLACE  long_name = "-200 TO -170 DEGC LOX Temp sensor"
   CONSTANT bypass_flag = byp_t7_object
      /* set the valves of objects to actual hardware ports here */
END DEFINE OBJECT {t7_object, t8_object}
/* object t8_object defined similar to t7_object */

DEFINE OBJECT r1_object IS
 low_tachometer_class
   REPLACE  long_name = "0 RPM to 3500 RPM Tachometer"
   CONSTANT bypass_flag = byp_r1_object
      /* set the valves of objects to actual hardware ports here */
END DEFINE OBJECT r1_object
```

```
DEFINE OBJECT {engage_clutch_p128_object,
               engage_valve_p128_object,
               engage_clutch_cool_object}
         IS control_point_class
   CONSTANT allowed_states = {'on', 'off'}
      /* set the valves of objects to actual hardware ports here */
END DEFINE OBJECT {engage_clutch_p128_object,
                   engage_valve_p128_object,
                   engage_clutch_cool_object}

DEFINE OBJECT motor_p128_speed_object IS control_point_class
   CONSTANT allowed_valves = {0 RPM, 3500 RPM}
      /* set the valves of objects to actual hardware ports here */
END DEFINE OBJECT motor_p128_speed_object

DEFINE OBJECT motor_p128_object IS {plus100_hp_motor_class,
                                    sslo2_pump_hwsubsystem_class}
   REPLACE  long_name = "Main fill pump motor"
   CONSTANT output    = clutch_p128_object
   CONSTANT control   = motor_p128_speed_object
   CONSTANT temp      = t1_object
   CONSTANT temp_byp  = byp_t1_object
END DEFINE OBJECT motor_p128_object

DEFINE OBJECT pump_p128_object IS {plus45_cm_pump_class,
                                   sslo2_pump_hwsubsystem_class,
                                   liquid_oxygen_precautions_class}
   REPLACE  long_name = "Main fill pump"
   CONSTANT input     = lox_storage_tank_object
   CONSTANT output    = valve_p128_object
   CONSTANT control   = clutch_p128_object
   CONSTANT temp      = t7_object
   CONSTANT temp_byp  = byp_t7_object
END DEFINE OBJECT pump_p128_object

DEFINE OBJECT clutch_p128_object IS {plus100_hp_clutch_class,
                                     sslo2_pump_hwsubsystem_class}
   REPLACE  long_name = "Main fill pump motor clutch"
   CONSTANT input     = motor_p128_object
   CONSTANT output    = pump_p128_object
   CONSTANT cooling_system = h2o_pump_p128_object
   CONSTANT control   = engage_clutch_p128_object
   CONSTANT temp      = {t2_object, t4_object,
                         t5_object, t6_object}
   CONSTANT temp_byp  = {byp_t2_object, byp_t4_object,
                         byp_t5_object, byp_t6_object}
END DEFINE OBJECT clutch_p128_object
```

```
DEFINE OBJECT  h2o_pump_p128_object IS
 {plus10_cm_pump_class, sslo2_pump_hwsubsystem_class}
   REPLACE  long_name = "Main fill pump clutch cooling H2O pump"
   CONSTANT input     = engage_clutch_p128_object
   CONSTANT control   = clutch_p128_cooling_object
   CONSTANT temp      = t2_object
   CONSTANT temp_byp  = byp_t2_object
END DEFINE OBJECT h2o_pump_p128_object

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

DEFINE OBJECT sslo2_alt_pump_hwsubsystem_object
                      IS sslo2_pump_hwsubsystem_class
   /* redefine all the instantiation specific information
      (i.e., ports, connections, ...) of the
      sslo2_pump_hwsubsystem_class */
END DEFINE OBJECT sslo2_alt_pump_hwsubsystem_object

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

DEFINE OBJECT fill_system_object IS hwsubsystem_class
   REPLACE  long_name = "ET LOX loading and monitoring hwsubsystem"
   CONSTANT contains        = {sslo2_pump_hwsubsystem_object,
                               sslo2_alt_pump_hwsubsystem_object,
                               sslo2_lox_tank_object}
   CONSTANT main_pump       = sslo2_pump_hwsubsystem_object
   CONSTANT alternate_pump  = sslo2_alt_pump_hwsubsystem_object

/* fill_system specific constants */
   CONSTANT slow_fill_time_limit     = 11 MINUTES
   CONSTANT fast_fill_time_limit     = 30 MINUTES
   CONSTANT complete_fill_time_limit = 10 MINUTES
END DEFINE OBJECT fill_system_object

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

$$$$$
```

APPENDIX E

LIST OF ACRONYMS

LIST OF ACRONYMS

ANSI	American National Standards Institute
ASCII	American Standard Code for Information Interchange
BNF	Backus-Naur Form
CDOL	Customer Data and Operations Language
CSTOL	Colorado System Test and Operations Language
DCL	Digital Command Language
DOS	Disk Operating System
ET	External Tank
ISO	International Organization for Standardization
JSC	Johnson Space Center
GOAL	Ground Operations and Aerospace Language
GSE	ground support equipment
GSFC	Goddard Space Flight Center
KSC	Kennedy Space Center
LL(1)	Left parsible with Look ahead of 1 symbol only
LO2	Liquid Oxygen
LOOPS	Lisp Object Oriented Programming System
LOX	Liquid Oxygen
LRU	Line Replaceable Units

LIST OF ACRONYMS (Continued)

NASA	National Aeronautics and Space Administration
SCOOPS	Scheme Object Oriented Programming System
SIMULA	Simulation Language
SSLO2	Space Shuttle Liquid Oxygen
SSME	Space Shuttle Main Engines
SSOL	Space Station Operations Language
STOL	System Test and Operations Language
TAE	Transportable Applications Executive
TCL	TAE Control Language
UIL	User Interface Language
USEWG	User Support Environment Working Group
VMS	Virtual Machine Operating System

LIST OF REFERENCES

1 Bongulielmi, A. P. and F. E. Cellier, On the Usefulness of Deterministic Grammars for Simulation Languages, Institute for Automatic Control, Zurich, Switzerland, 1983.

2 Wirth, N., Algorithms + Data Structures = Programs, Prentice-Hall, Series in Automatic Computation, Englewood Cliffs, New Jersey, 1976.

3 Cellier, F., R. Davis, C. Grim and C. Shaw, Specification for the Space Station User Interface Language (UIL) Version 1.0, Space Station User Support Environment Working Group (USEWG), 12 June 1989.

4 Ground Operations Aerospace Language (GOAL) Design, KSC-LPS-IB-070-1, Kennedy Space Center, Florida, 20 March 86.

5 Davis, R., "STOL, CSTOL, and the Space Station", University of Colorado, Boulder, Colorado, 1986.

6 TAE Command Language (TCL) - TAE Applications Programmer's Manual, Century Computing, 1988.

7 Space Station Operations Language (SSOL) System Level A Requirements Specification, KSC-DL-3032, Kennedy Space Center, Florida, 1986.

8 Customer Data and Operations Requirements (CDOL), Goddard Space Flight Center, Greenbelt, Maryland, December 1986.

9 CSTOL Reference Manual, University of Colorado, Boulder, Colorado, February 88.

10 Space Station Information User Support Environment Functional Requirements Document, JSC 30497, Johnson Space Center, Houston, Texas, 1987.

LIST OF REFERENCES (Continued)

11 Space Station User Interface Prototype, TRW System Development Division, Huntsville, Alabama, February 1988.

12 Reference Manual for the ADA Programming Language, ANSI/MIL-STD-1815A-1983, American National Standards Institute, 1983.

13 Stroustrup, B., The C++ Reference Manual, Addison-Wesley, Reading, Massachusetts, 1986.

14 Cox, B. J., Object Oriented Programming - An Evolutionary Approach, Addison-Wesley, Reading, Massachusetts, 1986.

15 Hewitt, C., P. Bishop, and R. Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence", Proceedings of the Third Joint Conference on Artificial Intelligence, 1973.

16 Weinreb, D. and D. Moon, "Flavors: Message Passing in the Lisp Machine", MIT Technical Memo Number 602, November 1980.

17 Bobrow, D.G. and M. Stefik, The Loops Manual, Technical Report KB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

18 Smith, J. D., An Introduction to Scheme, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

19 Bach, W. W., Is ADA Really an Object-Oriented Programming Language, Software Productivity Solutions, Inc., Indialantic, FL, October 1988.

20 Goldberg, A. and D. Robson, Smalltalk-80 - The Language and Its Implementation. Addison-Wesley, Reading, Massachusetts, 1983.

LIST OF REFERENCES (Continued)

21 Dahl, O.-J., E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press, London, 1972.

22 Dahl, O.-J., B. Myhrhaug and K. Nygaard, The SIMULA 67 Common Base Language, Norwegian Computing Centre, Forskningsveien 1B, Oslo 3, 1968.

23 Young, D. A., X Window Systems: Programming and Applications with Xt, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

24 Coded Character Set Standard ISO 646, International Organization for Standardization, Genera, Switzerland, 1988.