A Flexible Object Oriented Spacecraft Operating System (FOS)

Dave Ladouceur
Orbital Sciences Corporation
Space Technology Laboratory
3380 Mitchell Lane
Boulder, Colorado, 80301

Satellite Operating Software has traditionally been highly specialized custom software which operates one satellite according to deterministic rules. Software changes are usually accomplished with a complete reload from the ground, and performing patches require explicit knowledge of memory maps, variable locations, etc. and can often result in long term satellite down times. As computer, sensor and communication technology increases, more and more of the computing, routing and decision functions of small satellite systems are occurring on-board, and a need exists for adaptable flexible software. The object oriented approach to satellite operating systems provides a malleable system, resilient to failure, distributable across multiple satellites and easily adaptable to other applications. The Operating System acts as a switch for the distribution and execution of messages whether a command, code or data. Even operating system functions can only be executed by sending a message internally. These techniques provide for a safe system and simplified software maintenance. Since software code is broken into objects, the particular application can be distributed amongst one or more processors, satellites, ground stations or remote terminals. This allows for multi-processor based communication load balancing algorithms, dynamic fail-over capability and compute bound resource sharing. Since no explicit hardware knowledge is required by the flight application code, most objects can be reused for other satellite applications. Such systems can be implemented on small satellites using current processor technology.

INTRODUCTION

Most spacecraft whether large or small have one main purpose to gather and distribute data of some kind. Housekeeping including power management, attitude determination and correction, and other Telemetry, Tracking and Control functions have to take place to insure the health and capability of the spacecraft. Traditionally, this has been accomplished with analog and digital systems in a rigid deterministic fashion. The day to day operations of the spacecraft or sequences are loaded up to the flight computer on a periodic schedule. These sequences contain commands and time tags and are executed as specified until the next load. Until recently, software for each spacecraft was crafted in assembly language. Memory margins and CPU speed were of greatest concern and just making it work was a huge accomplishment. The previous generation of flight hardware has been replaced with high speed CMOS microprocessors and high density memories that remove the software limitations of the past, and can now provide a host for a robust flexible Flight Operating System.

OBJECT ORIENTED PROGRAMMING (OOP)

Object Oriented programming is a programming methodology that simplifies software development and maintenance. By definition many programs developed over the course of the last three decades were object oriented; they just did not know it. To be Object Oriented, the software must contain one or more of the following characteristics. Objects must exist in software that have attributes and exhibit behavior. These can be any physical piece of hardware or a real-world item that can be represented or modeled in software. An object is fine or course grained and represents related pieces of data and code. One object could represent a Global Positioning System receiver (course grained) or one object could represent a five line algorithm to average an array of numbers(fine grained). An object can belong to a class of objects in a hierarchy where an object at a lower level will inherit all the attributes (variables and definitions) and code of all higher classes. The structures, variables and procedures are dynamically made available for the programmer to use without declaration. This capability allows you to develop re-useable software by having general solutions at a higher level and then the capability to specialize a particular problem at a lower class level.

When an object is created, an instance(copy) of the object is made. This instance has attributes (local variables) that give the object its behavior. Each object has its own set of instance variables. The variables are completely contained within the object and can only be accessed by another object by sending the object a message.

Objects communicate with each other by sending messages, these messages can contain other objects, code or data. The destination object receives the message and takes the appropriate action based on the message content. Objects can be located anywhere. Each object can respond to the same message differently. A message may contain a command and that command can be responded to in a different way. This way, each application uses the same messages (commands) over and over again without knowing the

specific commands supported by the destination objects. If a command is not understood by an object, it returns a message that it did not know how to interpret the command. This allows you to develop a system incrementally. This ability for an object to respond uniquely to messages is called polymorphism in object oriented programming.

ADVANTAGES OF THE OOP INTERFACE

Safe Systems

Since objects can only communicate through messages, OOP promotes safe systems. Code and data are encapsulated in small chunks at the programmers discretion and can be made as safe, small grained and re-usable as practical.

Distributable Processing

The FOS provides the objects with a standard communication interface, this makes locality of an object immaterial. An object can exist on-board, at a Spacecraft Operations Center (SOC), in some remote groundstation or a user terminal. Therefore, a program can be executed partially on-board, on the ground or could be distributed amongst several satellites each contributing some data or code.

Parallel Processing

Multiple processors can be easily accommodated with FOS. Each processor can execute one or more objects in parallel with the other processors by sending messages through either a software or hardware communications interface.

Phased Development

The standard object interface also provides for the ability to migrate applications from the ground to onboard as they are developed. Experiments and one time operations can be run from the ground without uploading new flight code.

Flight Code Verification

Flight code simulations and testing are greatly simplified. Software drivers can be written to emulate any hardware component and respond with the appropriate messages through the communication interface. Since the applications are broken into small chunks, they can easily be replaced or upgraded with later revisions without concern of external dependencies. Only the messages that the object is required to respond to, have to be verified. All internal code is encapsulated and protected.

REAL TIME PROCESSING

Smalltalk, Common Lisp Object System (CLOS), C++, and Object Oriented Pascal are languages with built in Object Oriented constructs. Unfortunately, these fifth generation languages tend to require too much horsepower and memory to be an effective solution for small satellite real time processing. Therefore, the alternative is to implement the benefits of OOP techniques, namely *code re-usability, malleability and safeness* in a real-time communications environment. In order to perform high speed delivery of messages, the FOS messaging kernel behaves similarly to a PBX. It simply routes messages based on an objects address.

USERS

A User is defined as an object or chunk of memory that can respond, create or accept messages. Two types of Users can exist, synchronous and asynchronous. A synchronous User waits for a message and responds in a deterministic fashion. These Users are usually permanent (ie. telemetry frame builder). An asynchronous User is created dynamically as an instance of an object and has a lifespan as determined by the internal code executed by the object. These Users can be temporary or permanent(ie. a Fourier transform object).

PRE-DEFINED USERS

On most satellites several pre-defined users will exist such as transmitters, receivers and other intelligent hardware. These users will be at a fixed address. Other fixed Users will be defined as the software is developed.

ADDRESSING

Each User can belong to one or more classes of Users; in the FOS, the User/Class combination is made up of a sixteen bit address where ten bits are User (0..1023) and 6 bits are Class (0..63). This is arbitrary for each application. A modified X.25 protocol is used with a source and destination address, control and data fields. This protocol is used throughout the satellite(s), SOC and User Terminals. It is also used as the message passing protocol on board the spacecraft between objects(Users).

Source		Destination		Control	Information
User	Class	User	Class	as per X.25	as per X.25
10 Bits	6 Bits	10 Bits	6 Bits	8 Bits	0..255 Octets

Table 1. Message Specification.

QUEUES

Each User has associated with it one or more Queues. A queue is established for all incoming messages. Outgoing messages are placed in a Class Manager's Queue. A message consist of a queue block which contains the source, destination, control and information fields specified above. The information fields may contain data or code and is application specific. The queues can be self relative or absolute and have forward links (FLINK) and backward links (BLINK) associated with them. A Queue Head is associated with each Queue; the Queue Head has a pointer to the head of the queue (0 if empty), a count of the number of queue blocks in the queue, and an internal status word. A single user can control one or more queues as necessary to pipeline communications.

Queue Head	Type
Pointer to Head of Queue	Absolute Address
Queue Block Count	Unsigned Integer
Status	Unsigned Integer

Table 2. Queue Header for each User.

Queue Block	Type
Pointer to Next Element (0 if one element Queue)	Relative or Absolute Address
Pointer to Previous Element (0 if one element Queue)	Relative or Absolute Address
Internal Information	Implementation Dependent
Source Address	Actual Packet Starts Here Destination's (User 10 bits, Class 6 bits)
Destination Address	Source's (User 10 bits, Class 6 bits)
Control	(filled in by *protocol* Users)
Information	Data or Code to send

Table 3. Queue Block for Communication.

OPERATING SYSTEM CALLS

The FOS contains a kernel of prebuilt objects under Class (0), User (0..1023). These Users perform executive functions and provide services for the application Users (software) on board the satellite, at a SOC or in a remote terminal. Such services would include acquiring and disposing of queue blocks, timer functions, hibernation and multi-tasking requests. A basic generic set of services are provided to make development of the application software of the satellite simple and straight forward. Any high level language can interface to FOS; only three direct subroutine calls are provided to the programmer *Get_QBlock, Send_Message* and *Receive_Message*. All other functions are supported only through the communications interface.

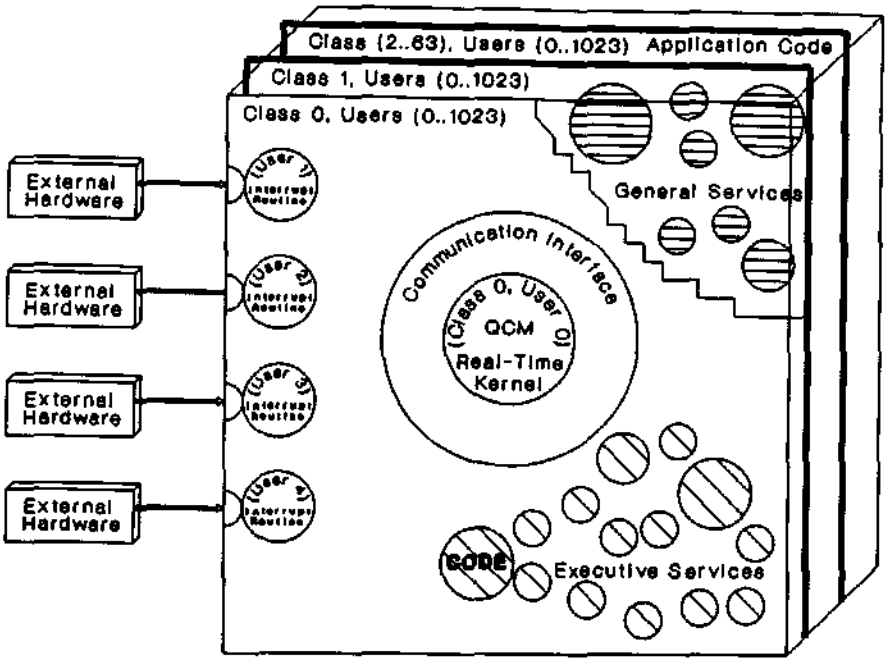

Figure 1. Object Oriented Flight Operating System Model

Send Message

Send_Message places an outgoing message into the Queue Class Manager (QCM) for the message's destination address class. Each Class (0..63) has associated with it a Queue Class Manager whose address is (CLASS #, User 0). The QCM is responsible for the delivery of messages for that class including messages to other satellites and the ground. If the message is destined for an on-board User, it is simply stuffed in the tail of the User's queue, and then the User is scheduled for execution.

Receive Message

Receive_Message takes the first message in the User's queue and returns a pointer to it. The User can then take the appropriate action based on the contents of the message.

APPLICATION CODE

Application software in a satellite is defined as any code that carries out the specific mission, whether a store and forward system, a sensor package or experiments. First, the application code for a Satellite is broken up into different classes of users. After each class of User is defined, then common generic functions and services (algorithms) are defined for all Users. This code could be added directly to the FOS executive as generic routines. Then specialized algorithms are split into asynchronous/synchronous Users and the software is developed to carry out the particular function in a high level language at the choice of the Application Programmer or Scientist. This development requires the programmer to break down his/her application into practical pieces that are distributed across several objects (Users). This method of problem abstraction is a natural extension to human problem solving and will greatly enhance the development effort[2].

PROCESSOR SUITABILITY

Almost any micro-processor can be used for this operating system. Processors with fast context switching and built-in high speed communication links would be greatly enhance total system throughput.

FUTURE DIRECTIONS

One of the problems with developing small code fragments is to get compilers to output relocatable code chunks without any dependencies on libraries, stack, heap and dynamic memory. To solve this, a fifth generation programming environment like Smalltalk that provides a generic language interface to sanitize standard C, Pascal or other 3rd generation language source code needs to be developed. This interface will compile or call the appropriate compiler and link the object code directly into the FOS.

SUMMARY

The Flexible Object Oriented Spacecraft Operating System can provide a building block by which software developed for space applications can be safe, malleable and re-useable. The flexibility of developing code piecemeal and testing it as you go, greatly enhances mission success. Proto-typing and simulations can be run using actual flight code. The ability to redistribute one or more pieces of an algorithm, greatly simplifies the development of experiments and applications that can not all be accomplished on-orbit. Another benefit of the system is simplicity; non-software engineering staff can successfully program flight code that operates under FOS. And of course, developing a *command sequencer* would be a trivial task.

Get_QBlock

Get_QBlock simply returns a pointer to a free queue block.

RUNTIME SYSTEM

Task Scheduler

The runtime system consists of a task scheduler, interrupt routines for hardware ports and timer related functions. The scheduler is a cooperative, queue-prioritized, round-robin scheduler. Basically when a user receives a message, it is placed in the scheduler process list. When a User executes a Receive_Message on an empty queue it is placed in an I-O wait state list. A task can avoid hibernation by checking its Queue Head to see if there is anything available for processing and perform low priority tasks until something is available. This is a permanent user acting in an asynchronous fashion. Active Users with empty queues are placed in a lower priority compute bound process list. However, each User can raise or lower its priority.

Re-Entrancy

In order to save memory and re-use as much code as possible, the FOS requires all code to be re-entrant and relocatable. The programmer must use queue blocks which are always relative to a single instance of a User object as the place to store local variables. No common stack operations are allowed due to the unpredictability of multiple high level languages accessing a common stack. Small work stacks are provided to each user as defined by the needs of the high level language. There are Executive Services for stack and queue operations in the traditional sense.

Threads

The creation of a message can create a task (execution thread) for each active User in the system based on the message type, either synchronous or asynchronous. Parallel programming techniques can be utilized if a multi-processor system was being used as in Transputers[1]. Each algorithm could be vectored out as appropriate to achieve satisfactory response.

REFERENCES

Smalltalk/V286 Tutorial and Programming Handbook, Digitalk Inc., 9841 Airport Boulevard, Los Angeles, California 90045, 1988.

G. Agha, *Architectures for Object Based Concurrent Programs*, Object Oriented Programming: Systems, Languages and Applications (OOPSLA), Conference Proceedings, 1989.

[1] - *Transputers - an Inmos Processor based on C.A.R. Hoare's concurrency model "Communicating Sequential Processes(CSP)"* (Hoare, 1978)

[2] - T. Winograd, F. Flores - *Understanding Computers and Cognition*, Addison-Wesley, 1987