Index: communication satellites, in-orbit testing, software, systems monitoring, transponders

Microwave measurement system software

K. D. FULLETT, W. D. KELLEY, V. E. RIGINOS, P-H. SHEN, AND S. L. TELLER

(Manuscript received December 9, 1992)

Abstract

To facilitate efficient, cost-effective development of in-orbit test (IOT) measurements and turnkey systems, microwave measurement system (MMS) software built on an engineered platform of reusable software services and facilities has been developed over the past several years and deployed in operational systems. The Measurement Processing and Control Platform (MPCP) provides modular software components that have been developed and tested for measurement scheduling, interprocess communications, and resource and system information sharing. Its code libraries support graphically based user interfaces, instrument control, instrument bus management, and error detection and reporting; and its data processing subsystems support database management, report generation, and interactive data analysis.

Operating in a network environment under a UNIX System V operating system, this multiuser, multitasking MMS software supports both local and wide-area networking, including remote access and control of the IOT measurement equipment. It executes in a distributed processing system architecture spanning a number of dissimilar workstations. With interprocess/intermachine communications provided by the MPCP mail system, the separate user interface and measurement programs can execute on different machines at different times, for improved operational flexibility.

This paper describes the design and implementation of the MMS software. The concepts and methods presented are also applicable to other measurement-oriented systems (such as those used for communications system monitoring), where escalating software costs must be controlled.

Introduction

Computer-controlled in-orbit test (IOT) systems integrate microwave measurement equipment, computer hardware, and measurement software into a unified test facility for measuring the communications subsystem performance of an orbiting satellite. To assess performance, the IOT system conducts a variety of microwave frequency measurements. These include the measurement of spacecraft input power flux density (IPFD) and equivalent isotropically radiated power (EIRP); transponder frequency response; gain transfer; group delay; gain-to-noise temperature ratio, G/T; and others [1],[2].

The development of modern IOT systems is traced by Shen *et al.* [3]. As noted in that paper, IOT is performed for acceptance testing immediately following launch; to monitor communications subsystem performance throughout the satellite's operational lifetime; and to investigate anomalies. The specific missions determine the IOT system's basic requirements. The fact that newer satellites contain more transponders and have increased payload complexity compared to earlier generations places greater demands and constraints on the IOT systems built to test them. Because the satellite owner desires to place the satellite into revenue-generating operational service as soon as possible after launch, the IOT system is constrained to accomplish its task as quickly as possible, especially during acceptance testing. Increased satellite capacity and complexity have also resulted in greater volumes of test data, which must be maintained and reported. Finally, the pool of spacecraft experts available for performing complex IOTs and evaluating the data is distributed more thinly as the number of satellite networks in service increases.

In addition to its primary mission of performing IOT measurements, the modern computer-controlled IOT system must address such system requirements as real-time and network operation, human-machine interaction, and remote access and control of the measurement hardware. The system must provide a user interface that is easy to use, yet flexible enough to accommodate the various IOT missions. The capability to execute measurements concurrently and to support a multiuser environment are desirable system features.

These requirements and constraints place greater responsibility for IOT system functionality on the software, which must address the network, hardware, and communications environment of a distributed processing system spanning a number of workstations. As a result, the IOT software, with its volume, sophistication, complexity, and difficulty of development and control, has come to dominate both overall system cost and scheduling. The development of custom software is time-consuming and costly, and requires highly skilled personnel. Methods and techniques are continually being sought to make the

IOT software production process more efficient. By contrast, the cost and scheduling aspects of hardware implementation for IOT and similar measurement systems are generally well-understood and well-controlled.

This paper discusses the principal software concepts underlying development of the Measurement Processing and Control Platform (MPCP), which served as the foundation for design and implementation of the microwave measurement system (MMS) software. Specific applications for computer-controlled IOT measurements and turnkey systems are addressed. A companion paper [4] describes the implementation of the MPCP software in a specific IOT system.

The principles underlying a robust, software-engineered platform such as MPCP are also applicable to the software implementation of similar measurement-oriented, computer-controlled systems, such as communications monitoring systems. Like computer-controlled IOT systems, these systems require escalating amounts of software for which costs, scheduling, and quality must be controlled.

Software development methodology

IOT systems are uniquely designed to test specific satellite characteristics and networks; however, many IOT measurement subtasks are the same from one IOT system to the next. Such commonality of function underlies much of the MMS design.

Because IOT systems are unique and custom-built, the software implementation of computer-controlled IOT systems is not standardized, nor is there a standard IOT software architecture. Riginos *et al.* [5] contrasts two approaches to designing such software. In one approach, measurements are implemented one at a time in a self-contained manner. Each measurement performs all required functions, including instrument control, user interface, and data processing tasks such as database management, printing, and plotting. As new measurements are required, an existing measurement is copied and modified to meet the specific requirements. While this self-contained approach has certain attributes, such as moderate levels of developmental effort for succeeding measurements, its use for large-scale systems software development also contributes to problems in terms of life cycle maintenance, quality, capability, flexibility, and extensibility.

Early implementations of computer-controlled IOT systems employed the self-contained measurement methodology. After several such systems had been implemented, it was realized that 80 to 90 percent of the measurement tasks—such as managing the user interface, controlling and managing the

instruments, managing data and files, reporting and logging errors, plotting and printing output data—were common to all IOT measurements. This realization formed the basis for a fundamentally different strategy of building IOT systems and measurements.

This alternative approach, based on a software-engineered platform of reusable software components, was the one selected for developing the MPCP and MMS described here. Although the initial design and development effort is substantial, the engineered platform implementation results in improved software characteristics in terms of life cycle (reusability, maintainability, expandability, portability, and evolution), quality (methodology, robustness, consistency, and flexibility), and capability (remote control, networking, concurrent measurements, distributed systems, and user-driven changes).

Well-conceived and well-implemented reusable software components can significantly decrease the scheduling and performance risks associated with large-scale software development. With MPCP components as a base, new IOT systems and measurements can be implemented in a cost-effective and timely manner.

The MPCP operating system

The MPCP is a special-purpose operating system that provides an integrated platform of facilities, subsystems, and services to the measurement application program. These include interprocess/intermachine mail communications, measurement scheduling and resource management, a system-wide shared-data depository called the datapool, standardized file management, database management, alarm management, and printing and plotting. Object code libraries are provided for instrument control, IEEE-488 bus control, uniform error handling, user interface support facilities, and other utility functions that can be linked with applications such as IOT measurements.

The MPCP operating system is implemented via the UNIX System V operating system, enhanced with Berkeley sockets for communication and IEEE-488 bus control functions for measurement equipment interfacing. Because MPCP is implemented in the C and C++ languages, it executes with high run-time efficiency and is highly portable to other machines.

The basic concept is to design and implement task-specific modules that can be independently tested, refined, and expanded. Although the modules are functionally specialized, a major design goal is generality within the problem domain of the specific function. Over a period of several years, MPCP was developed as a platform of IOT system and measurement code building blocks that could be reused between measurements and across systems. The use of pre-tested modules substantially reduces the development time, cost, and per-

formance risk of unproven software. Because of the functional similarity of IOT systems and measurements, the code reuse percentage for MPCP is quite high.

MPCP design goals

MPCP implementation follows generally accepted software engineering principles, practices, and open system standards. These are briefly described below, with emphasis on why certain design choices were made, rather than on how the software is specifically implemented.

Because customers require remote access and control of an IOT system, an important design goal was to support a networked, distributed-processing hardware environment. The MPCP executes in such an environment. A typical IOT system architecture is shown in Figure 1.

Another principal design goal was to separate functional tasks into dedicated processes that could execute in distributed-processing, networked environments. A "process" is a program that is being executed in the host machine. Each process performs a specialized task with well-defined external interfaces. For example, an IOT measurement inputs parameters from the user, manages the hardware during data acquisition, saves the data in a database, and prints or plots the measured data. The overall measurement is implemented as two separate processes: one that interfaces with the user, and another that manages the measurement equipment and performs the actual measurement. The IOT system's scheduler, datapool, and earth station management facilities are implemented as self-contained dedicated "daemon" processes (*i.e.*, processes executing continuously in the background of the UNIX operating system). The printing and plotting tasks are similarly managed. Problems are generally contained within a specific process, and task-specific programs can be modified and recompiled when necessary, with little or no impact on the interfaces.

With separately executing processes, interprocess communication is required. This capability is provided by the MPCP mail subsystem, which client processes can access via calls to a library of mail functions. The mechanisms used by the mail subsystem are completely transparent to clients. To support a distributed processing (multiple-host) execution environment, the mail subsystem facilitates intermachine interprocess communications in which processes can execute on different host workstations and computers connected via a transmission control protocol/internet protocol (TCP/IP) network. Figure 2 depicts such an arrangement, with each box representing a separate workstation or computer.

Following another software principle, the MPCP is structured in a top-down hierarchy which permits the software to be partitioned according to function, and distinguishes between high-level and low-level functions. An

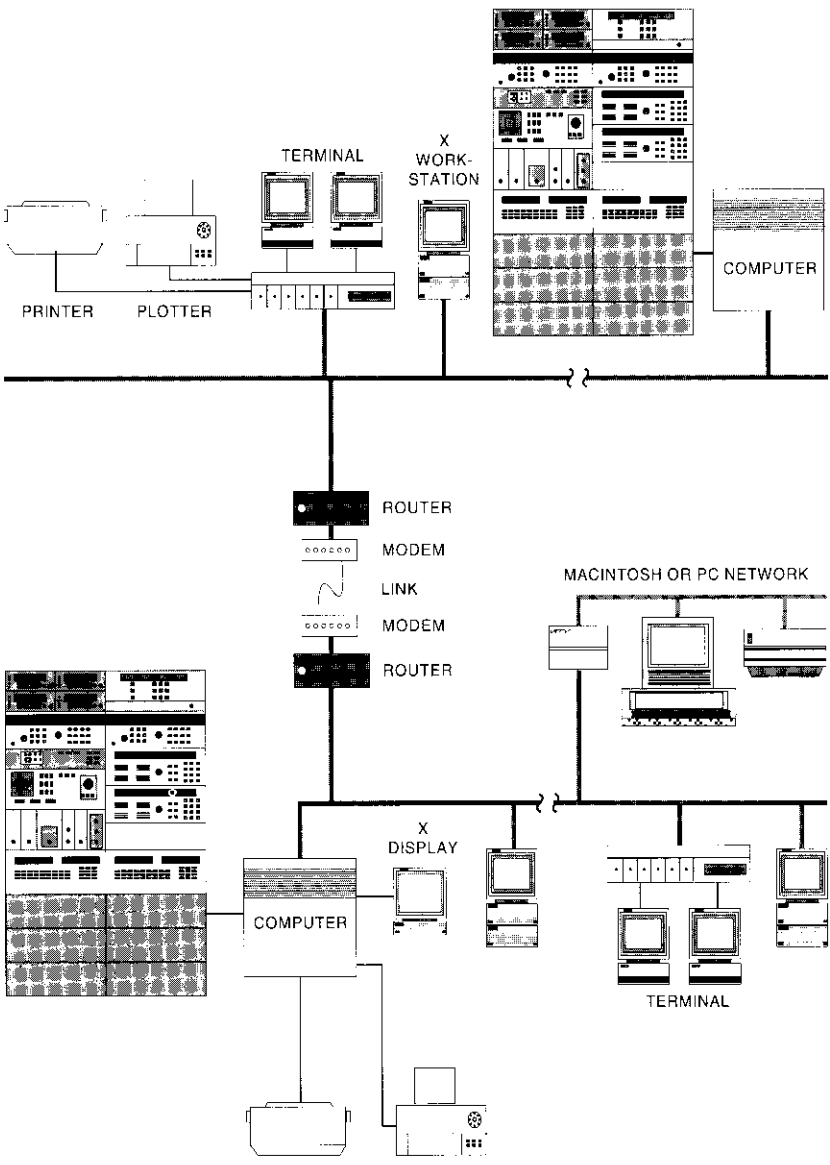


Figure 1. *Networked System Architecture*

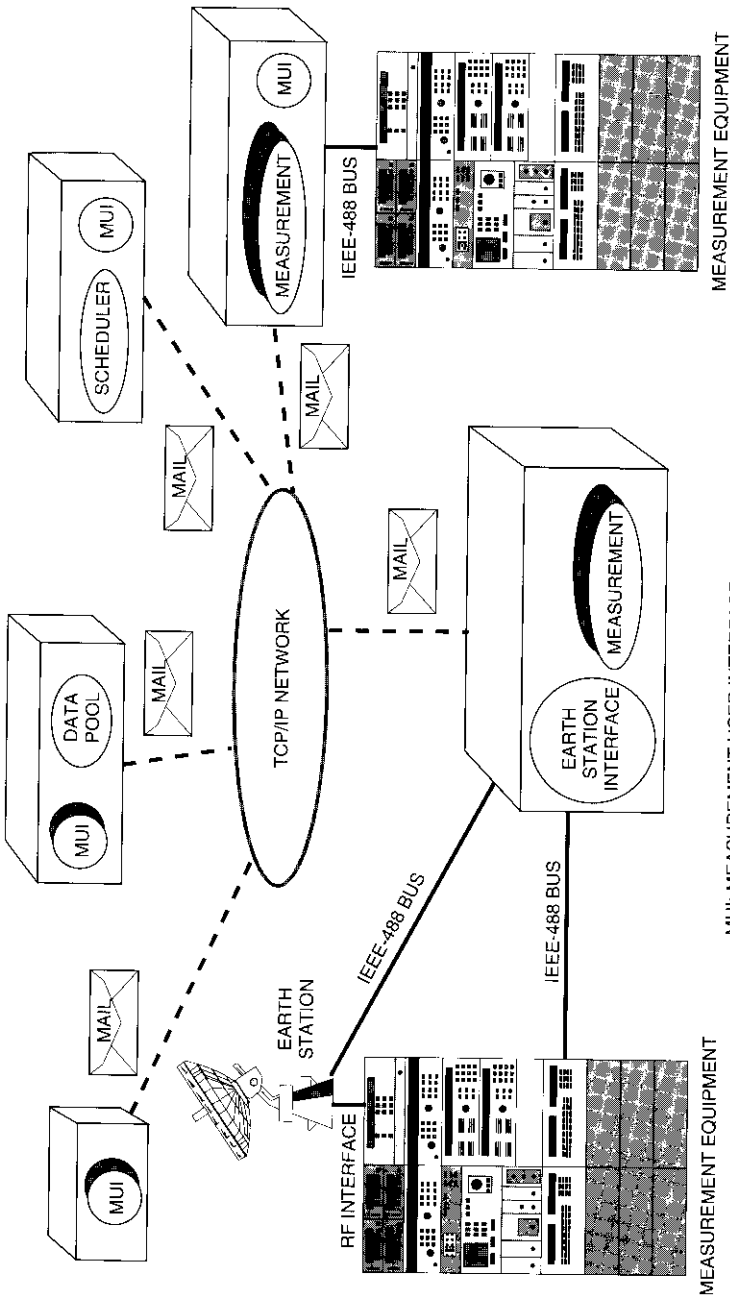


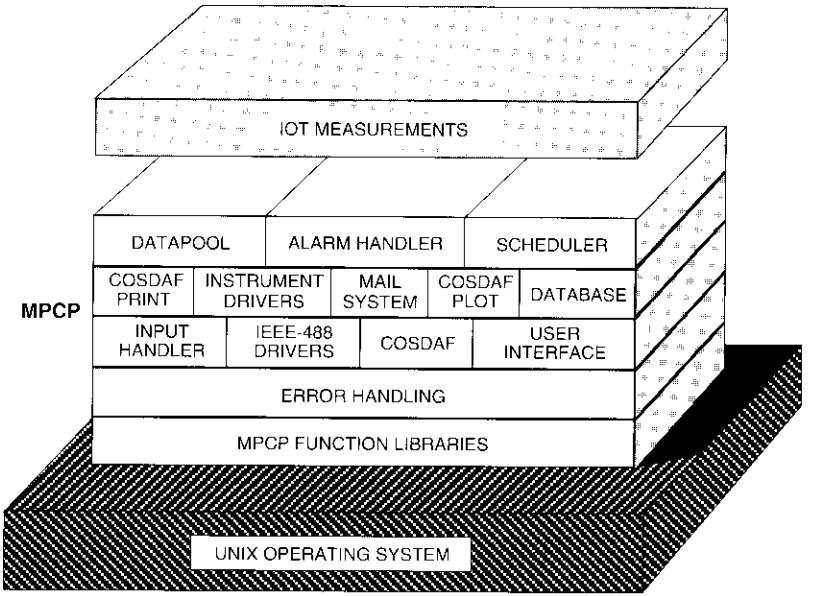
Figure 2. *Distributed Processing Execution Environment*

IOT measurement program, for example, is a high-level task. With proper code structuring, the software developer who implements a high-level measurement program need only be concerned with orchestrating the logical sequence of activities necessary to perform the measurement, and not with the details of lower-level functions. While the measurement may be required to access instruments, the details of instrument management are left to the driver that controls each instrument. Similarly, while the instrument driver exerts control via messages sent across the IEEE-488 bus that connects the instrument to the computer, the developer of the instrument driver need not be concerned with the details of managing the bus. Instead, the developer has available an IEEE-488 library of bus management functions. MPCP services such as the scheduler and datapool are conceptually at a level below that of the IOT measurement, but above lower-level functions such as the 488 library. At any level, the software developer has available the building blocks of lower-level MPCP facilities, and can access them through well-defined interfaces. Figure 3 illustrates the general nature of this hierarchical implementation of MPCP.

An important objective in building computer-controlled IOT systems is to maintain data integrity by preventing corruption [5]. "Defensive code" is used to detect error conditions and trap them before they propagate through to corrupt the measurement data. Each module performs extensive error checking of its inputs, as well as on the results of its own processing. Error-trapping is performed by all levels of code. When errors are detected at any level, they are managed consistently by calls to an error handling library, which underlies all upper levels of code, as depicted in Figure 3. Error detection and reporting throughout the code allows effective tracing of both programming errors and operational errors (*e.g.*, a disconnected instrument). Error handling techniques are discussed later in this paper.

The MPCP Function Libraries also extend under all upper levels of code. These libraries are used to segregate low-level functions from higher levels, and are generally accessible by any program, although access to functions is usually through the hierarchy of code modules.

As in hardware system design, a large software system such as an IOT or measurement system is more easily managed by decomposing it into smaller functional components, which are then treated as individual subsystems. Subsystems are self-standing software projects which encompass a group of similar and related functions that can be specified and implemented individually. Changes in one subsystem are generally localized and do not usually affect other subsystems. Decomposition of a large software project promotes modular system design, with the result that the project is easier to design, implement, and enhance.



COSDAF: COMSAT Data File Format

Figure 3. *MPCP Operating System*

To achieve the design goals of maximum code reusability and system implementation flexibility, the concept of data-driven design is employed throughout the MMS. Whenever possible, the data that programs use are separated into static ASCII text files that are human-readable and easily edited. The program's behavior can be altered by simply editing the data file, rather than changing the program code that processes it. This results in programs that are easier to develop, test, and maintain, and that have the flexibility to accommodate varying requirements from one IOT system to the next.

The user interface is designed for ease of use and flexibility, and also has the ability to check for erroneous user input (as far as practicable). The interface is implemented separately from the measurement program, and is called the measurement-user interface (MUI). The user interacts with the MUI via an X Window to specify IOT measurement parameters and scheduling information. The MUI display format is controlled by input data files, which are easily modified by simple editing of ASCII-encoded text files, without the need for program code changes. The user is notified if out-of-range data values are entered. To reduce keying, the MUI is initially displayed with default values preloaded in all fields and parameters.

Because each task is specialized, the module's developer focuses on the specific problem at hand and optimizes the code for that task. With well-defined, constant interfaces, modules can be modified or improved without affecting other modules. New modules, such as new instrument drivers, can readily be added to the system, often by using an existing driver as a template and modifying it as necessary.

Systems applications

Two systems are described which demonstrate the code reusability of the MPCP software building blocks. Although these systems differ substantially in mission and requirements, the MPCP platform provided the software foundation upon which the application-specific code was built.

In the first example, an IOT system using MPCP building blocks was designed, implemented, and deployed for the European Telecommunications Satellite Organization (EUTELSAT) [1],[2]. This system has performed the IOT of four EUTELSAT II spacecraft, and continues to monitor their performance.

In the second example, MPCP components were used to implement RF terminal supervisory equipment for a National Aeronautics and Space Administration (NASA) ground station for the Advanced Communications Technology Satellite (ACTS) program. In this system, a network of three engineering workstations, supporting three simultaneous operators, performs supervisory, status, and control functions for terminal and ground station RF equipment.

Communications and network environment

The IOT system architecture supports local and wide-area networking (LAN/WAN), as well as several communications protocols, network structures, and transmission media. The system's transport/network layer implements TCP/IP to guarantee end-to-end data delivery and integrity between communicating devices on networks that support this protocol. The LAN's physical layer implements the IEEE 802.3 (CSMA/CD Ethernet) protocol on coaxial cable operating at 10 Mbit/s. The LAN connects to workstations, displays, terminal servers, peripherals, communications equipment, and other PC-based LANs, as shown in Figure 1. The system supports WAN across leased lines, public switched telephone networks, and public data networks at rates ranging from 2.4 to 19.2 kbit/s, with link-limited transmission speed. Standard serial communications via RS-232 protocols and modem-connected data links are also supported. As illustrated in Figure 1, a complement of microwave measure-

ment equipment is connected to the IOT system workstation at the host earth station via the IEEE-488 digital instrumentation standard bus.

This networked system architecture provides for equipment and resource sharing, operational flexibility, performance enhancement, incremental redundancy, and vendor and hardware independence. System resources such as measurement hardware, plotters, printers, and communications facilities are shared among IOT measurements and users. Incremental expansion of computing, storage, display, and communications devices, as well as additional peripherals, are readily accommodated. Additional workstations can be connected to the network for load-sharing, and processes can execute on different machines for operational flexibility and improved performance.

An important consideration when implementing a network-oriented system is the ability to support interprocess and intermachine communications. The X Window protocol supports communications in a network of dissimilar but X-compatible workstations. When processes reside on the same machine, Berkeley sockets support interprocess communications. The MPCP mail subsystem also uses Berkeley sockets to implement a higher-level mechanism for interprocess communications that extends across machine boundaries.

MPCP mail interprocess communications

The MPCP mail subsystem provides clients with high-level, reliable, and easy-to-program facilities for interprocess, intermachine, and internetwork communications. The subsystem does the following:

- Provides interprocess communications (using TCP/IP) for different processes operating on separate workstations across the network.
- Provides "atomic" transmission (data treated as an indivisible unit) of large data structures.
- Provides the sender with verification of transmission.
- Implements a client-server model.
- Notifies clients if a connection is lost to a server process.

The MPCP mail subsystem supports the first three features in a fashion similar to its main paradigm: the postal system. A client process mails data to another process at a given address. The various data items to be mailed are enclosed in an "envelope," which is received and/or delivered at the same time. Like the postal mail system, the mail subsystem handles all delivery details. Atomic transmission ensures that input/output transactions, once started, are completed without interruption. When the addressee receives an envelope,

the sender is provided with a "return receipt" (acknowledgment) verifying successful transmission.

The MPCP mail subsystem software implements the last two features based on a second paradigm: an open telephone line between a client and a server. Using the mail subsystem, a client (*i.e.*, a process that requires some service) establishes an open line with a server (the process providing the service, such as the scheduler or datapool). Once the line is established, mail can be exchanged between client and server. If the server process terminates or there are problems in the network, the line is disconnected. The client process detects the disconnection and reports an error condition. Any number of clients can be supported in the network. For example, all MUI processes are served by the scheduler. There can be any number of servers of different kinds in the network, but there is only one server of a particular kind, such as the MPCP scheduler.

Process-to-process mail communications are supported when processes execute on different machines connected to the network. This feature permits a distributed processing system implementation in which there may be more than one machine with multiple clients and servers hosted on different machines, as depicted in Figure 4. In the figure, S1, S2, S3, and S4 represent different kinds of server processes, such as the scheduler, datapool, alarm manager, and earth station interface manager. The lines connecting clients to servers are mail connections, which are supported across serial data links.

The MPCP scheduler

The MPCP scheduler is a server process that provides scheduling and resource allocation across the network. The scheduler executes in the UNIX system background as an independent daemon process and accepts requests for jobs and resources from other processes. Scheduling is non-preemptive and is provided on a first-come, first-served basis. Jobs are run based on the requested time and the availability of resources. The scheduler exchanges messages with its clients via the mail subsystem, and receives requests from MUI processes to schedule a measurement process at a specified time. The scheduler manages the sharing of resources, handles global remote/local control of instrumentation, and provides the means for a user to determine the status of a job that is running in the background.

Measurements can be scheduled by the user to run at any time of day, on any day. They can also be scheduled to run repetitively for a user-specified duration at a user-specified interval. Multiple measurements can be scheduled for execution at any time. The scheduler also supplies the link necessary to

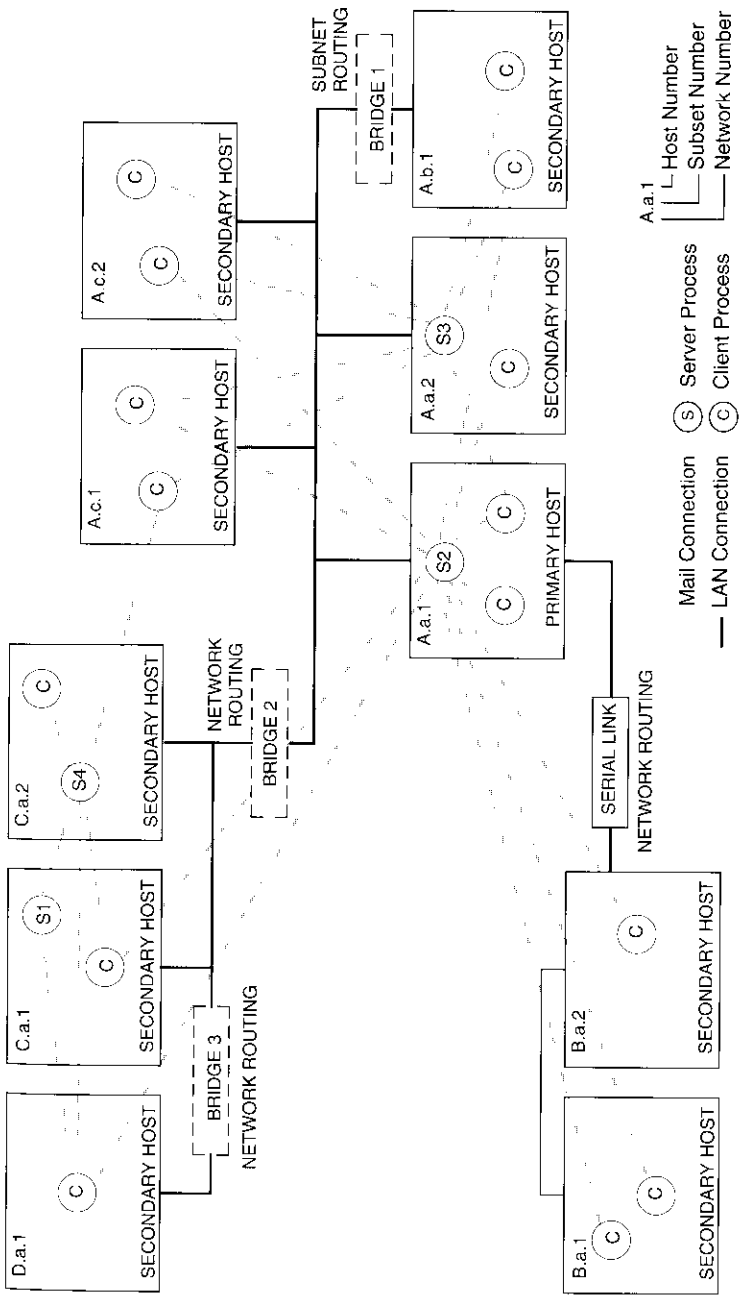


Figure 4. *MPCP Distributed Processing Mail System*

communicate scheduling information between the MUI and the measurement processes.

When the time arrives to execute the requested measurement, the scheduler verifies the availability of the required resources and initiates the measurement process in the appropriate workstation. When the measurement process begins, it requests from the scheduler (via mail) the needed resources. The scheduler determines that a mail connection has been established with the measurement process, and that the process is actually executing. It then grants the requested resources and locks them for the duration of the measurement, or until the mail connection between the scheduler and the measurement process is broken. By ensuring that the process is actually running before resources are committed, the scheduler prevents a potential lockup situation. If the scheduler were to begin a measurement process and immediately allocate and lock the resources, the measurement could fail to start for some reason, or fail to establish a mail connection, and the resources would be unavailable for other uses.

If the resources required for a scheduled measurement are unavailable, execution of the measurement is deferred until they become available. Since scheduling conflicts are resolved on a first-come, first-served basis, a particular measurement may have to wait its turn in a job queue. A planned expansion of the scheduler will provide for the prioritization of measurements.

An example will illustrate the scheduler's operation and interaction with measurement processes. A system comprising three workstations, with one spectrum analyzer and two power meters connected to workstation 3, is assumed. MUI programs in workstations 1 and 2 have each requested that an EIRP measurement be performed at 10:00 a.m. the next morning. In addition, workstation 2 has requested a power measurement at the same time. EIRP measurements require both a spectrum analyzer and a power meter, while power measurements require only a single power meter.

At 10:00 a.m. the following morning, the scheduler ascertains that both power meters and the spectrum analyzer are available, and grants the request from workstation 1 (which was received first) by executing an EIRP measurement and locking power meter 1 and the spectrum analyzer. As a result, the EIRP measurement request from workstation 2 cannot be granted at this time, since only power meter 2 is available. Since the power measurement also requested by workstation 2 requires a single power meter, the scheduler initiates the power measurement process, establishes a mail connection with that process, and locks power meter 2. When the first EIRP measurement terminates and both the spectrum analyzer and power meter 1 become available, the scheduler executes the EIRP measurement requested by the MUI process running on workstation 2, establishes an open mail connection with the EIRP

measurement process, and locks power meter 1 and the spectrum analyzer for the duration of that measurement. When the measurement process terminates (normally or abnormally), the scheduler releases the resources, which are then available for the next request.

The scheduler determines when to start a particular measurement, based on the system clock and the scheduling information provided by the MUI when the measurement was specified. The UNIX operating system could start jobs at the scheduled time, if that were the extent of the requirement. However, the scheduler performs two additional, essential functions. First, it begins a job at the scheduled time *with the user-specified arguments*, which can vary in number and value with each running of a measurement.

The second, more fundamental function of the scheduler is to manage the sharing of one set of microwave measurement and earth station equipment resources. These resources include microwave test equipment (such as the spectrum analyzer and power meters), earth station equipment (such as uplink and downlink chains, the antenna, automatic saturation control units, and radiometers), files, mail connections, and memory. Each resource is identified by name (resource ID) and the maximum number of users. Resources may also belong to resource groups, in which case the group is given a single name, such as "Uplink 1," which would include the entire chain of earth station equipment forming an uplink. The scheduler can allocate both individual resources and resource groups to jobs. Resource sharing is cooperative, not enforced or preemptive.

The scheduler also manages the state of resources when they are not controlled by a measurement process. For example, it makes certain that instrumentation used by a job is placed into a quiescent state when a job is complete. This ensures that jobs cancelled abortively do not leave instrument resources in an undesirable or unknown state. Since the scheduler controls all system resources, it is responsible for handling user requests to place instrumentation into its local state when it is not being used by an executing process. Such requests from measurement processes are handled via the MPCP mail subsystem.

The MPCP datapool

The MPCP datapool server is a memory-resident data area that functions as a depository for information to be shared system-wide. All TOT system processes can access the datapool, which will accommodate arbitrary data. The datapool implements a client-server model in which clients such as MUI processes update entries in the datapool and/or request the most up-to-date information from the datapool, via the mail subsystem.

Maintaining data integrity within the datapool and preventing "racing" conditions are critical design issues. A racing condition can arise as follows. Suppose process A reads the current value for datum 1 in the datapool and, based on that information, updates datum 2 in the datapool. Process B has previously read datum 2, prior to its update by process A, and, based on this information, updates datum 1. Process A assumes that the datum 1 value it read is valid and up-to-date, when in fact datum 1 was subsequently updated by process B. Thus, process A continues execution with data that are not current. Similarly, process B assumes that the datum 2 value it read is valid and up-to-date, when in fact datum 2 was subsequently updated by process A. At this point, a racing condition has been created in the datapool, and neither process has up-to-date data. This situation is avoided by implementing the datapool as described below.

The datapool is not merely a passive receptacle and reporter of data, because it can notify clients of changes to datapool entries. A client can register with the datapool a list of entries of interest. Whenever the status of a list entry changes (*e.g.*, it is changed in value or deleted, or the process that owns the entry terminates), the client is notified by the datapool and provided with the current value for the data item. A client may request and receive the current value of any item in the datapool at any time, and thus is assured of having the most up-to-date values for items of interest.

As shown in Figure 5, client processes do not access the datapool directly. When they need to add, modify, or remove data, clients access the datapool via a library of datapool functions. These functions then transmit the request to the datapool process, which accesses the data area on behalf of the clients. Only the datapool process can access entries in the data area itself, to prevent potential corruption of the datapool by client processes and to maintain datapool integrity. A data-locking mechanism is applied to guarantee that only one data update can occur at any given time.

System-wide standard messages are used to transmit information to and from the datapool. A typical datapool process operation involves two types of message transmission: the client's request message received by the datapool process, and the acknowledgment returned with the requested data to the client. All communications between the datapool and its clients (Figure 5) are via the MPCP mail subsystem, which is transparent to clients. Datapool functions are available to the client to request data creation, deletion, modification, retrieval, and the addition or deletion of the client process from the notification list. These functions assemble mail messages, send mail to the datapool process via the mail subsystem, and inform the calling program whether or not the operation was successful. The datapool library implementation hides both the

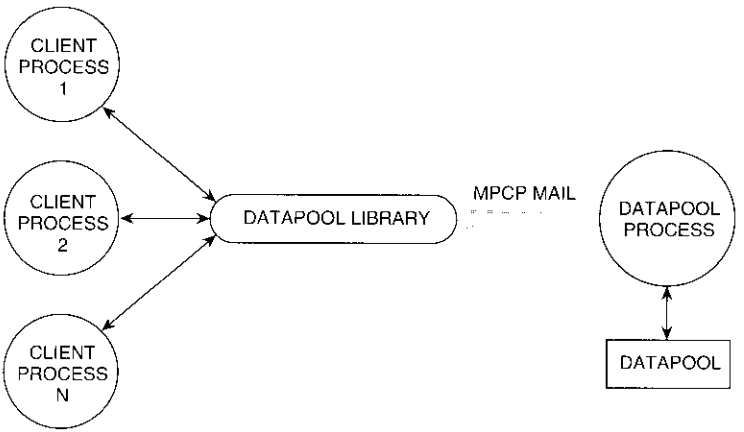


Figure 5. *MPCP Datapool and Clients*

type of interprocess communications media and the fact that messages are used.

The datapool also supports the operation of the MUIs. When an MUI is opened by a user, it establishes a mail connection with the datapool. Assume, for example, that client process 1 in Figure 5 is an MUI in which the user has specified spacecraft 1, and client process 2 is another MUI in which spacecraft 2 is specified. If a third MUI is opened, it may want to know toward which spacecraft the earth station is currently pointing, or if another process changes the current spacecraft in the datapool. The datapool maintains such system-level information and can notify a requesting client process of the current status and configuration for both the spacecraft and the earth station.

IOT measurement implementation

IOT measurements are the core of computer-controlled IOT systems. An IOT system is composed of numerous IOT measurement programs, their corresponding MUIs, and system control and data processing functions. This section discusses the concepts and principles underlying the implementation of measurements. Overall software organization, data-driven design, user interface implementation, error handling, and data processing features are described.

Since measurement data are a primary concern in an IOT system, data integrity is essential. Because the measurement is often fully automated, the measurement code must be able to detect and report errors, which can arise

from many sources, including real-world anomalies. For example, a measurement may require the use of a piece of test equipment that has been turned off or disconnected. The code then issues an error message, and the user may correct the problem (*e.g.*, reconnect the instrument) and request that the measurement continue, or cancel the measurement.

Measurement architecture

The ideal IOT is one in which the overall measurement is implemented as two separate programs: the MUI, and the measurement itself. A model of the IOT measurement architecture is shown in Figure 6.

Communication between the MUI and the measurement processes is facilitated by the MPCP mail subsystem, the scheduler, and the datapool server. The mail subsystem provides interprocess communications between the MUI and the scheduler, and between the scheduler and the measurement program or other peer client processes. Properly designed interfaces between communicating processes are essential for coordinating the various activities associated with the overall measurement.

The scheduler allocates system resources, including earth station resources, and maintains their availability status. The MUI sends a job request message via the mail system to the scheduler and communicates the name of the measurement program, the name of a file containing a list of measurement arguments (called the "argsfile"), and scheduling information. Before starting a job that invokes a measurement program, the scheduler determines the availability of required resources by accessing a file that lists the resources required by each measurement. If the resources are currently unavailable, the scheduler places the job in a job-wait queue and reschedules it.

If the resources are available, the scheduler starts the measurement via a command to the UNIX operating system. If UNIX initiation is successful, the program becomes an executing process. The measurement process performs an initialization and establishes a mail session connection to the scheduler. It then accesses the resources file and sends the scheduler a message indicating that the process is initialized and running, and requesting access to system resources such as the spectrum analyzer, an uplink path in the earth station, and an uplink synthesizer. If available, the resources are allocated as requested.

By structuring the overall measurement in this manner, MUI and measurement programs can be designed, implemented, tested, and maintained independently and in parallel. This supports modularity and encapsulation of the respective programs. Team personnel with complementary skills can work on different tasks, making optimum use of their software capabilities and

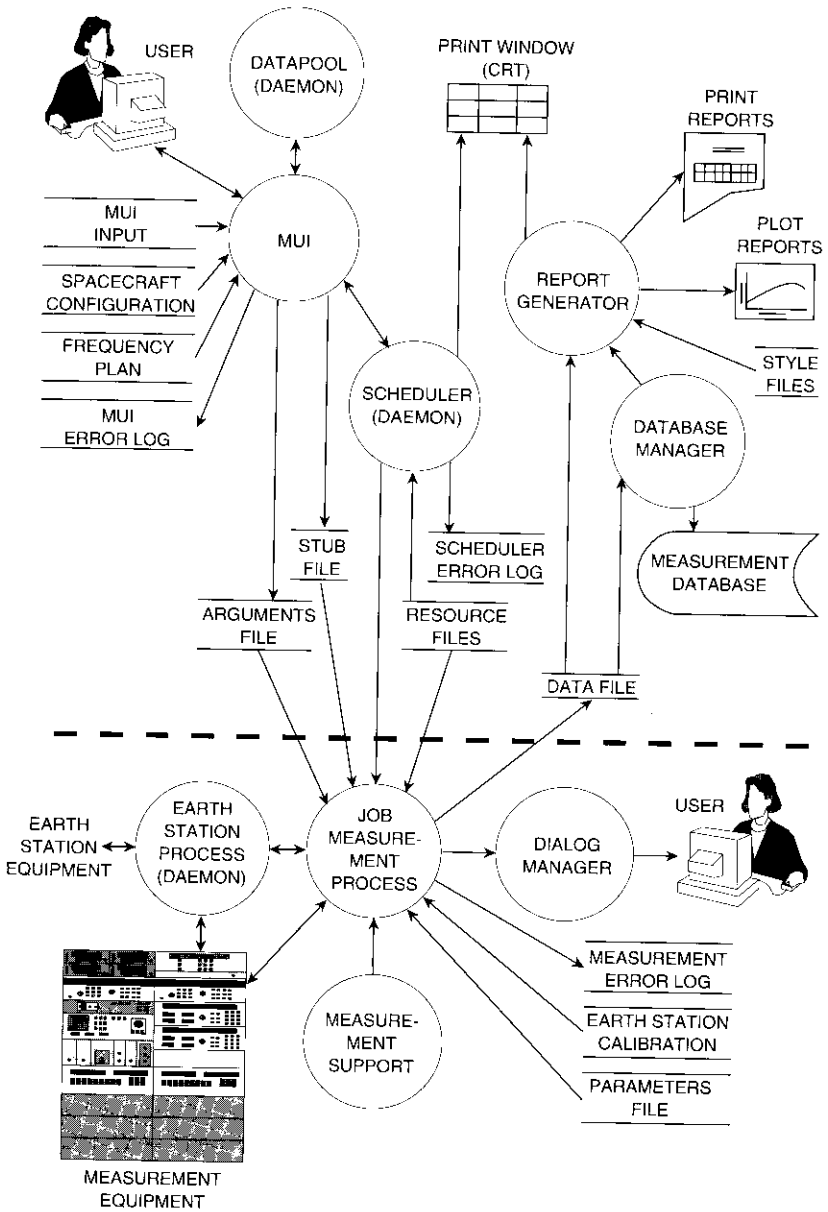


Figure 6. *IOT Measurement Model*

experience. Each program developer uses the most appropriate programming paradigm. For example, since the user interacts with the MUI via a mouse and keyboard, the MUI must respond to unpredictable events (*e.g.*, a mouse click or the pressing of a key) and is therefore implemented using an event-oriented programming paradigm. The measurement program, on the other hand, interacts with the microwave measurement and earth station equipment in a predictable manner, and thus is implemented in an algorithmic, procedure-oriented programming paradigm (although it can be interrupted by unexpected behavior when an instrument issues a service request interrupt on the IEEE-488 bus).

Data-driven implementation

The MMS software is implemented by using text files and avoiding hard-coding wherever possible. As shown in Figure 6, text files are used to exchange information between processes, as a complement to interprocess mail communications. The following files are used, and will be discussed in context: MUI input, measurement argument, stub, resource, spacecraft configuration, earth station calibration data, measurement parameter, and print and plot style. Since these files are structured during the system design phase, they can be constructed to hold any information desired, and can be customized to meet customer requirements. Thus, the files provide flexibility and adaptability to the measurement system design.

Both the file management procedures and storage format are standardized within the MMS software. File operations such as open, read, write, and close are performed via calls to an MPCP library. Files are formatted in a standard COMSAT Data File Format (COSDAF) and stored as ASCII-coded text. COSDAF files can be viewed, edited, and imported into other application programs.

System behavior can be altered by editing the files, which minimizes the need to recompile the program when changes are required. For example, the appearance of an MUI window on the display is controlled by an MUI input file. The number of MUI controls, their position, and type (*e.g.*, pushbuttons, edit fields, and pop-up menu selections) are easily changed by editing this file.

Although MMS files are easily edited, they are static in the sense that they generally remain unchanged for a particular sequence of IOT measurements. In fact, once constructed, they seldom change, although they can be modified when necessary. The spacecraft configuration file, for example, includes information regarding spacecraft characteristics, channel characteristics such as center frequency and bandwidth, transmit and receive beams, transponder gain settings, and orbital parameters. Another file, the frequency plan, identifies channels and/or carrier slots, their center frequencies and bandwidths, power threshold levels for alarms, expected signal modulation, and other frequency-

related information. Other files, such as the earth station calibration file, store calibration information such as antenna gain *vs* frequency and coupler value *vs* frequency.

The measurement process reads particular files when required. The use of common files by different measurement processes guarantees uniformity of information. Also, since the user does not have to constantly reenter the same information through the keyboard, user input errors and fatigue are reduced.

User interface design

Effective operation of a measurement system is highly dependent on the design, behavior, flexibility, ease of use, and consistency of the user interface. More than 30 years of human-machine interface research (References 6 and 7, for example) have indicated that the most effective technique is a graphically based interface that allows a user to indicate a desired action by "pointing and clicking" in windows on the display by using a mouse device. The keyboard is used to enter parameter values and data. This technique is in contrast to the older command-line-based interface that requires the user to accurately remember and type in esoteric command codes. Graphically based interfaces are now common on many computer systems, such as the popular Apple Macintosh [8] and desktop computers running Microsoft Windows. The X Window-based MUI enables the IOT system user to specify measurement parameters. It contains edit fields, pushbuttons, toggle buttons, and radio buttons. Buttons are actuated by pointing and clicking with the mouse. A typical MUI is shown in Figure 7.

MUIs and other windows meet the following IOT system operational requirements:

- Permit the user to easily, quickly, and intuitively set up IOT measurements.
- Enable the user to search through the measurement database to retrieve files meeting user-specified criteria.
- Enable the user to process high volumes of measurement data into plots and printouts.
- Inform the user of input range and type errors.
- Inform the system operator of errors encountered during measurements and other activities.
- Require minimal training, so new users can gain proficiency rapidly.
- Preserve operational flexibility for nonroutine activities, such as anomaly investigations.

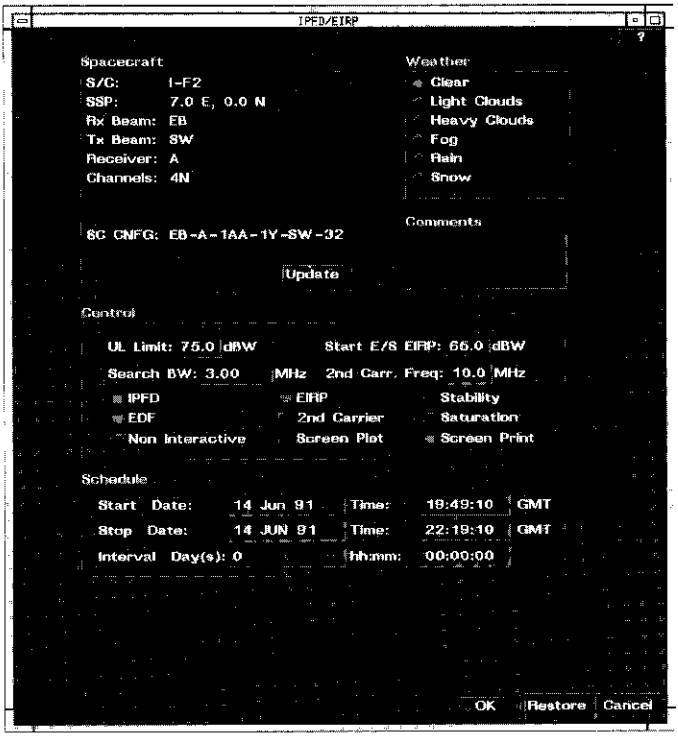


Figure 7. *FLUX/EIRP Measurement User Interface Window*

Some IOT measurements can be run automatically, without the presence of an operator, while others are interactive and require operator inputs throughout the measurement. Some measurements can be run in either mode (selectable by the user as the "Non Interactive" option shown in Figure 7), which can be toggled on or off by clicking the mouse.

Implementation of an effective user interface requires careful thought and considerable effort. All MUIs and other windows are implemented using Open Software Foundation's OSF/Motif toolkit and style guidelines [7]. MUIs are implemented to be consistent in behavior and similar in appearance. Buttons and controls that perform the same function from one MUI to another are positioned in the same location, so that the user who has learned one MUI has a familiar model to follow. When opened, MUIs are displayed with default parameter values and control settings, and have a "form-fill-in"/menu-selection presentation format. The default values are read from an ASCII file, which is

easily modified. If an input is invalid (*e.g.*, an out-of-range value is typed in), the user is immediately notified of the error and prompted for another input.

Often, several input parameters are coupled in a dependency relationship to preserve maximum operational flexibility. If a user specifies a parameter that is coupled to others, the related parameter or parameters will also be changed automatically. For example, the user may be required to specify a bandwidth, a step size, and the number of steps to be performed by a measurement. These parameters may be coupled such that specifying the number of steps and step size automatically determines the bandwidth parameter as their product.

MUI-measurement process interface

Once the user has configured the measurement via the MUI, the specified parameters and controls are communicated to the measurement process via a command-line interface similar to the standard UNIX system command interface. In normal operation, the user initiates a particular measurement by filling in the appropriate MUI and scheduling the measurement. The MUI is displayed on a workstation or X-Terminal. The user then presses the OK button on the MUI (see Figure 7), and the MUI communicates this information to the measurement process via the scheduler. At the scheduled time, the scheduler checks to see if the required resources are available, starts the measurement process, establishes a mail connection to the process, and allocates the requested resources.

To preserve maximum system flexibility, measurement programs can be run without an MUI workstation or X-Terminal by using a standard character-based ASCII terminal. A user can run an IOT measurement at the host earth station from a remote site, such as the user's home, by using a personal or portable computer and modem. For example, during an anomaly investigation, it may be desirable to alter the normal flow of system operation or to make a particular series of measurements not implemented by the IOT system MUIs. This flexibility is achieved as described below.

Following UNIX conventions, an IOT measurement program can be invoked from a connected terminal by typing the name of the executable program and optional arguments as follows:

```
meas_name [-opt <opt_arg> ...]
```

where meas_name is the name of the measurement program, -opt specifies an option, and <opt_arg> specifies an argument to an option. For example, the command line to invoke the IPFD/EIRP measurement with options set for a 3-MHz search bandwidth and saturation is

```
flux_eirp -Search_bw 3.0 -Saturation
```

The IPFD/EIRP measurement program is invoked and instructed to set the search bandwidth on the spectrum analyzer to 3.0 MHz and perform the measurement at saturation. The arguments available to each IOT measurement are customized for that particular measurement. Any control available in the MUI can be entered as a command-line argument.

A measurement typically requires numerous controls and parameter settings. Normally, the MUI handles all controls; however, the number of controls can present a problem for manual entry via a command line. Since the command is transitory, whenever a measurement is repeated (perhaps with different options), the measurement command and its arguments must be reentered—a tedious and error-prone process. One solution is to place measurement arguments into an arguments file. Once constructed, this file is permanent, regardless of the options stored. The argsfile can be configured with a set of default options.

To accommodate the use of the argsfile, the measurement program extends the conventional UNIX command-line invocation with one additional argument -args <argsfile>. The -args option instructs the measurement process to obtain its command-line arguments from the file named in the parameter, <argsfile>. If an argsfile has been created for the IPFD/EIRP measurement, the measurement program can be invoked from the UNIX command line as follows:

```
flux_eirp -args my_args
```

where the file my_args contains the arguments -Search_bw 3.0 and -Saturation. Once again, a data-driven design approach is used to preserve maximum operational flexibility and adaptability.

The ability of the measurement program to receive its arguments from an argsfile provides the necessary interface between the MUI and the measurement program. When the MUI window (*e.g.*, Figure 7) is opened, the parameters are displayed with default settings which the user is free to alter to accommodate a specific measurement configuration. When the measurement specification is complete, the user presses OK. The MUI program then creates an argsfile containing the specified arguments and parameters for use by the measurement process (as depicted in Figure 6). Use of the argsfile minimizes the number of entries required from the user, since most of the defaults are unchanged.

Measurement program implementation

The measurement program focuses on the measurement task itself. For example, although the measurement process (a program in execution) outputs a data file of results, it is not responsible for storing, displaying, printing, or plotting the data. These tasks are managed by other processes. Similarly, the measurement program is decoupled from the particular spacecraft upon which

it will perform the measurement, and spacecraft information is communicated from the MUI program via the argsfile.

Measurement programs are implemented in a modular manner using object-oriented design and implementation techniques [9]. This results in a high degree of encapsulation. Using the class terminology of object-oriented programming, a generic "Measurement" class is implemented with attributes (such as data declarations, data structures, and methods) that are common to all IOT measurements. Measurement-class methods include performing initialization and establishing a mail session with the scheduler, opening files, checking arguments passed via the argsfile for validity, configuring uplink and downlink equipment, performing up- and downlink measurements, storing the final output data, pausing the measurement, and cancelling the measurement when requested by the user. Since these common tasks comprise the bulk of the measurement, it is appropriate to aggregate them into a generic class.

Measurement programs are coded in C++, a language designed to support object-oriented programming and the construction of classes of objects with inheritance relationships. Figure 8 illustrates the class structuring and inheritance relationships of some of the IOT measurements. The implementation of measurement programs as classes of objects allows the developer to "leverage" code using inheritance, as explained below. This enhances reliability because each software element is tested thoroughly every time it is leveraged and reused. As a result, greater emphasis can be placed on measurement technique, rather than measurement mechanics.

Inheritance enables the measurement developer to leverage code as follows. Using the inheritance properties of C++ [10], an EIRP measurement is implemented as a subclass (or derived class) of the generic Measurement class. The EIRP measurement inherits all of the data structures and methods of the "parent" Measurement class, and implements additional structures and methods as well.

The gain transfer, in-band frequency response, and spurious output measurements are subclasses derived from the EIRP class. Gain transfer is an EIRP measurement performed at different uplink power levels of a test signal; frequency response is an EIRP measurement performed at different frequencies in a spacecraft transponder channel; and spurious output is an EIRP measurement performed on a specific spurious signal received from the communications satellite. These measurements inherit the data structures and methods from the parent EIRP class, as well as from the "grandparent" Measurement class, with additional structures and methods implemented to accommodate measurement-specific requirements. Thus, the developer of the gain transfer measurement can focus on issues unique to that particular measurement, while reusing or

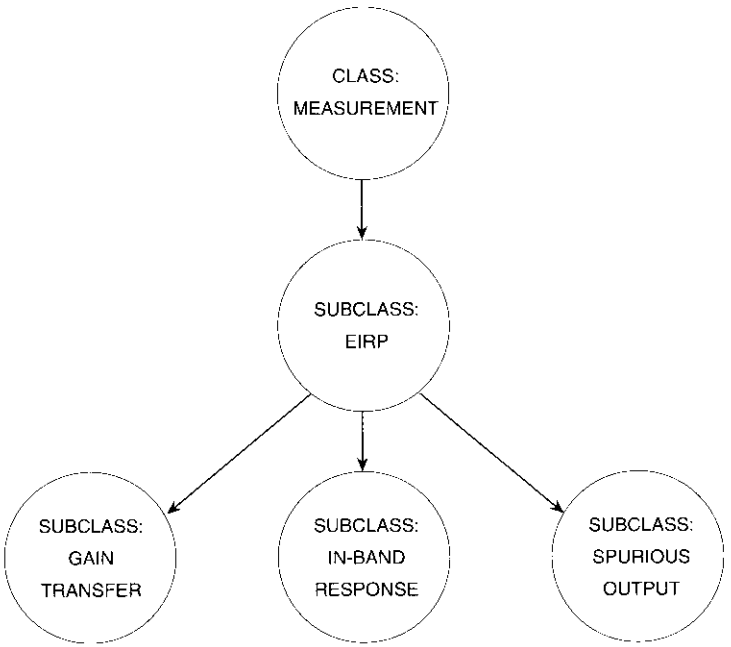


Figure 8. *IOT Measurement Class Structure*

"leveraging" the previously tested code for the EIRP and Measurement classes from which it is derived. This repeated testing improves overall software reliability.

Class-structured, object-oriented implementation of measurement procedures enhances code flexibility and adaptability. For example, the gain transfer measurement program can be modified without affecting other measurement programs, such as in-band frequency response. On the other hand, global changes can be made by modifying and recompiling the generic Measurement class. Its derived classes inherit the modifications upon recompilation.

An IOT measurement is constructed as a hierarchy of modularized code layers. The main program is linked with various libraries to create the executable program. These libraries include application-level libraries (*e.g.*, Measurement and EIRP class libraries); MPCP libraries (*e.g.*, measurement support, instrument drivers, IEEE-488 bus management, mathematical, and utility functions, and error management); and UNIX system libraries (*e.g.*, device input/output). The typical layered code organization of the in-band frequency response measurement program is illustrated in Figure 9.

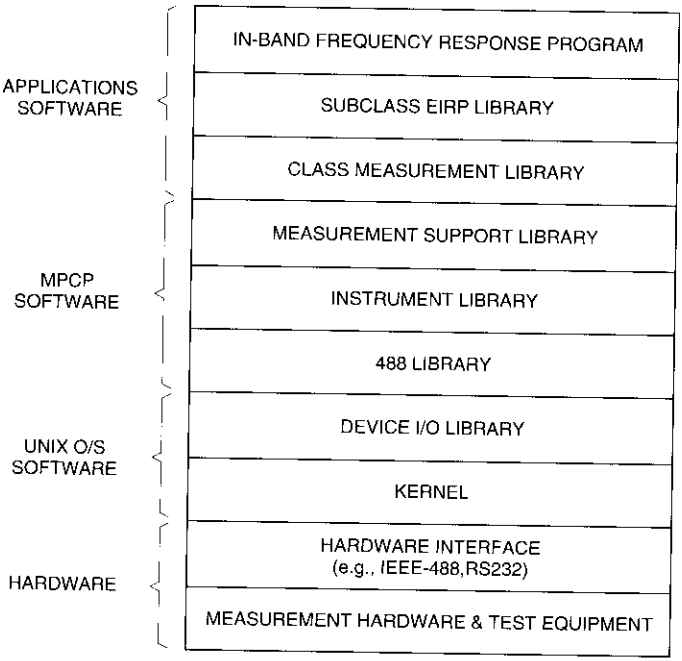


Figure 9. *Layered Measurement Program Code*

The main program makes a sequence of calls to functions in the EIRP and Measurement class libraries. These, in turn, call functions in the MPCP measurement support library, which perform tasks required by all measurements, such as initializing hardware, creating and opening files required by the measurement program, and performing housekeeping and cleanup activities.

Instruments are accessed by measurement support library functions via drivers in an instrument library. Instrument drivers perform high-level operations such as initializing the instrument, changing its settings, and reading the measured results. Each driver contains functions for its particular instrument. Drivers have been implemented for:

- Spectrum analyzers
- RF and waveform synthesizers
- Network analyzers
- Modulation analyzers
- Noise analyzers

- Frequency counters
- Power meters
- Voltmeters
- RF switch controllers
- Data acquisition units.

The driver provides a simple interface with the software developer using the instrument. Because data integrity is a principal concern, the driver is capable of detecting and reporting errors that may arise from interactions with the instrument, and handles the instrument's software control capabilities. This enables the developer to focus on instrument usage, rather than on the detailed mechanics of communicating with the instrument.

Instrument drivers communicate with instruments via the IEEE-488 bus by using the MPCP 488 library, which provides high-level bus management function calls to the driver. The library performs atomic I/O with a given instrument, so that an I/O transaction, once started, cannot be interrupted. This allows the same instrument to be shared by multiple processes.

The 488 library also performs such functions as sending an Interface Clear signal to all instruments, writing a command string to an instrument, reading a response string from an instrument, addressing an instrument to talk or listen, retrieving the bus and address of an instrument when multiple buses are present, and performing a serial poll of an instrument to obtain its status byte. These functions are generally called by the instrument drivers, but can also be called by other programs for communication with other IEEE-488-compatible devices such as computers or special-purpose hardware. The 488 library frees its user from bus management details.

Functions in the 488 library call low-level, primitive functions supplied with the UNIX operating system device I/O library. These primitives interact with the UNIX kernel (which directly controls the IEEE-488 bus) through an interface card. A cable connects the workstation's interface card with the instruments, which are daisy-chained onto the IEEE-488 bus through cables.

An instrument may be used in the shared mode in which it is checked out from the scheduler, or may be managed completely by a device process. In the shared mode, the instrument (if available) is checked out from the scheduler as described previously. The measurement is not allowed to run if any of the required instrumentation or other resources are unavailable. In the second case, the instrument is fully managed by a device process. All requests are issued to the instrument in the form of messages to the device process, which then communicates with the particular device or instrument. The device is not always a measurement instrument, but may be another computer that controls and communicates with such earth station equipment as the uplink power

meter, radiometer, or antenna control unit. In general, the shared mode is employed for instruments that are used occasionally by some measurements, while the device process is employed for instruments used by all measurements (*e.g.*, to communicate with a separate earth station control computer).

Other MPCP libraries, including utility libraries, a mathematics library, and the error handling library, are also linked to the measurement program. Several utility libraries provide useful functions required by IOT measurements or by other libraries, such as calculating EIRP, flux density, path loss, spacecraft gain, spreading factor, and slant range. The math library contains numerous mathematical functions routinely required in IOT and other measurement systems, such as numerical integration and linear regression analysis. The error handling library, which is used universally by all components of the software system, is described below.

Measurement programs execute as processes to perform actual measurements. In object-oriented terminology, a measurement process is an "instantiation" of the measurement; that is, it is an instance of a measurement program executing with parameters specified by the user.

A measurement process accesses several files, as illustrated in Figure 6. When the user specifies and schedules a measurement in an MUI window, the MUI process creates an argsfile and a stubfile. The stubfile contains annotations for the specific measurement (*e.g.*, the spacecraft selected, earth station name, weather conditions, comments, and the date and time) which do not affect the measurement itself. The measurement process obtains arguments from the argsfile and opens the stubfile, to which it appends measurement data. The process reads the resource file for the resources it requires, and the earth station calibration file for earth station antenna and coupler calibration data. The process may also access a parameters file for measurement-specific parameters.

Also, during measurement execution, the measurement process configures and controls the microwave measurement equipment for data acquisition, and communicates with the scheduler. It interacts with the earth station daemon process to obtain relevant information, and issues dialog windows to the user via a dialog manager process. Real-time measurement data are displayed on the workstation monitor as the measurement progresses (as depicted in Figure 6), along with the status of the measurement. The final measurement data file is stored in the database, printed, and plotted.

Error subsystem design

Error detection and management are critical considerations in the design and implementation of a computer-controlled IOT system or similar

measurement-oriented system. The MMS software is implemented to detect as many errors as possible, in order to prevent data corruption.

Two broad categories of errors can be manifested. Operational errors occur when the measurement process detects some condition or circumstance coded in the program to be an error. For example, an operational error occurs when an instrument required by a measurement cannot be initialized because it is powered-off or disconnected from the computer, or a printer is off-line or out of paper. If an operational error cannot be corrected, the measurement process must be aborted. Other errors are manifested as programming errors. It is important to detect both types of errors in a timely fashion and to obtain as much information as possible concerning the circumstances that gave rise to the error, to assist in diagnosis and correction.

The MPCP error library contains functions for assembling logging, and displaying error messages. When an error is detected, the specific function name and line number at which the error occurred are recorded.

Each layer of code is implemented to detect and report errors occurring at that level. Because of the layered software architecture, error reporting is stacked into a composite error message, as shown in Figure 10. When an error is detected, an IOT measurement program at the highest code level [*e.g.*, f1( )] places a message into a stack. Each lower layer of code [f2( ), f3( ), and f4( )] then places its own message onto the stack, so that the composite error message depicts the full path from the application program, down through successive layers, to the lowest-level function [f4( )] in which the error was detected. Figure 11 depicts a sample of a composite error message. The complete error diagnostic, showing the full path through the various layers of code, provides a context and clues for those tasked with troubleshooting and correcting the error. Error files can be printed in hardcopy form for later examination and analysis.

Measurement output files and data processing subsystems

If an IOT measurement process executes successfully to completion, it produces a measurement data file. These files are formatted as standard COSDAF files, and, because they are ASCII-encoded, can be viewed, edited, and imported into other applications, such as word processing or graphics software. Once data files have been produced, they can be processed by other MPCP subsystems for database entry, printing, and plotting. Permanent measurement data files are stored in a database.

The MPCP Database Management subsystem stores, searches, and retrieves files for display, printing, and plotting, as depicted in Figure 6. The subsystem

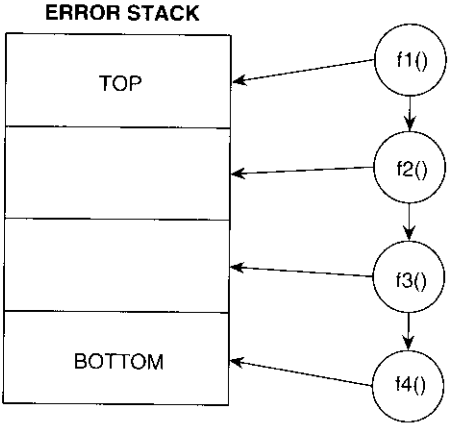


Figure 10. *Error-Reporting Stack Mechanism*

Mon May 17 14:11:34 1993 /proj/eutel/bin/flxerp

flxerp measurement cannot measure flux/eirp for channel 4.

meas_init_hdwr(): Unable to initialize hardware.

ms_init_hdwr(): hp3488a_new failed.

_hp3488a_init(): ZZ401 switch unit - slot #1 unavailable.

_hp3488a_get_slot(): ZZ401 switch unit - cannot communicate with switch control unit.

_hpib_io_dcderr(): HP-IB timeout, receive data ZZ401 9 /dev/hpib/1.

System call hpib_io(): Connection timed out (errno is 238).

Try Again Cancel

Figure 11. *Composite Error Message Format*

consists of a formal database containing a summary of the information associated with every measurement that has been stored (called the "index"), and a set of separate measurement data files containing the raw and processed results from each measurement (called "files"). The index is used to quickly locate data files of interest, just as a card catalog is used in a library. Once a particular data file has been located, it may be plotted, printed, or post-processed in some fashion.

The MPCP Plotting Services subsystem plots measurement data both during and after the measurement. The subsystem supports measurement systems in which many different types of data plots, to either soft output (*e.g.*, CRT displays) or hardcopy devices, are necessary for real-time measurements and post-measurement data analysis, as depicted in Figure 6. Plot formatting is specified by a style file that can easily be edited to change the appearance of the plot, without changing either the measurement data or the plotting program. Figure 12 illustrates specifications that can be accommodated in the plot style file.

The MPCP Printing Services subsystem prints measurement data, supports systems output, and formats data in much the same way as the plotting subsystem. Printouts are generated automatically at the conclusion of a measurement process, or at user request.

The MPCP Interactive Plotting subsystem significantly extends the post-measurement data analysis and manipulation capability of the measurement system beyond that supplied by MPCP Plotting Services, and provides a general-purpose capability to prepare finished, report-quality plots and graphs. It can plot any pair of columns of data in a COSDAF file. This is significant because, although the measurement data file contains a table with many columns of data, the MPCP Plotting Services subsystem normally plots only two columns (or three columns: Y1 and Y2 *vs* X). The Interactive Plotting subsystem also supports graphs with two *y* axes (Y1 and Y2 *vs* X). The data on a plot can be manipulated and edited in the following ways:

- Points can be cut and pasted.
- Scales, axes, and labels can be changed.
- Graphs or points can be annotated and/or marked.
- The plot can be zoomed in or out.
- New data points can be added via the keyboard and/or from existing files.
- Data from one or more files can be plotted on the same graph as data from another file.

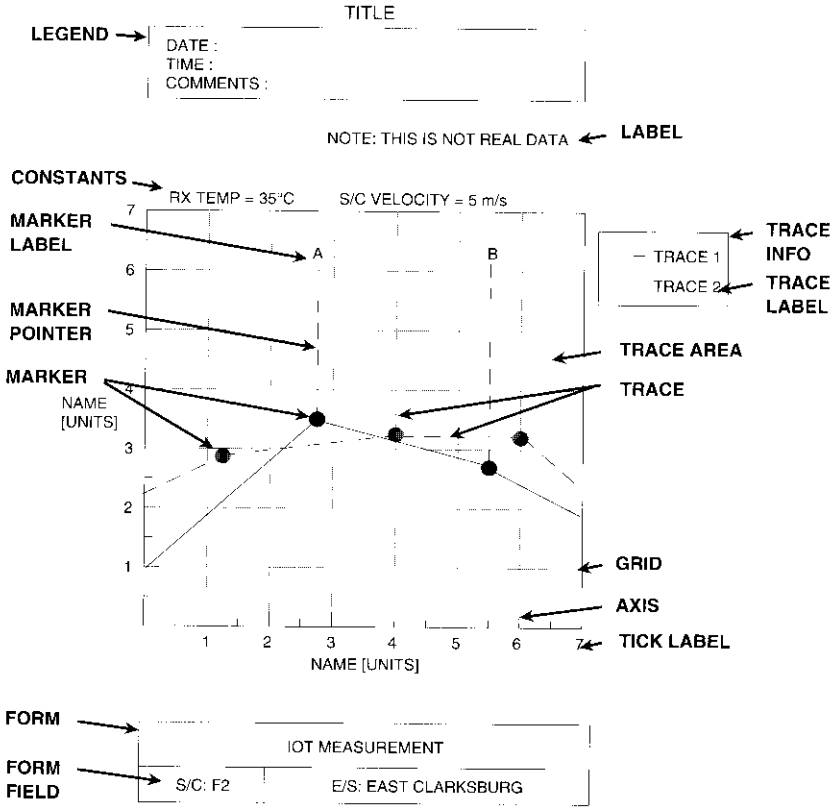


Figure 12. *MPCP Plotting Specifications*

Various data transformations are also supported. Two or more traces can be merged, algebraically added, or algebraically subtracted. For example, a calibration file can be subtracted from a measurement file, or two measurement traces can be subtracted from one another, leaving the residual differences. Finished plots can be stored and later retrieved.

Conclusions

The concept of building MMS software on an engineered platform of reusable facilities and services has been presented. Contemporary software engineering principles and practices—such as design-for-reusability; modularity and encapsulation of task-specific functions; object-oriented methodology and

implementation; hierarchical layering of code; linkable object code libraries; and processing subsystems—were used to develop and deploy robust, flexible, adaptable MMS software.

The desire for code reusability directed development of the MPCP, a special-purpose operating system that significantly reduces the time and expense involved in developing and implementing a cost-effective MMS. The field-tested MPCP software enables the limited number of expert software developers available to focus on the application's design and implementation, rather than on the software infrastructure required to support modern IOT and other microwave measurement and control systems.

Supported by the MPCP interprocess/intermachine mail communications subsystem, scheduler, and datapool, the IOT measurement architecture physically and logically partitions the overall task into separate user interface and measurement program entities, each optimized to perform a specific task. The network system architecture enables the user interface program and measurement program to execute on different machines and at different times of day—providing the system with a high degree of operational flexibility, including remote access and control of the IOT measurement equipment.

The concepts and methods applied in this study to the complex task of building automated IOT systems in a dynamic environment are also applicable to similar measurement-oriented systems, such as those used for communications systems monitoring.

References

[1] Y. Tharaud, B. Kasstan, and P. Barthmann, "IOT System for the EUTELSAT II Satellites," Global Satellite Communications Symposium, Nanjing, China, May 1991, *Proc.*, pp. 168–177.

[2] Y. Tharaud and V. Riginos, "EUTELSAT's Facilities for Measurement of Earth Stations and In-Orbit Satellite Payloads," 23rd General Assembly of the International Union of Radio Science (URSI), Prague, Czechoslovakia, August–September 1990.

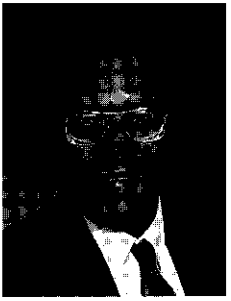
[3] P-H. Shen, V. Riginos, and S. Bangara, "In-Orbit Testing of Communications Satellites: The State of the Art," Global Satellite Communications Symposium, Nanjing, China, May 1991, *Proc.*, pp. 150–159.

[4] K. D. Fullett *et al.*, "The EUTELSAT In-Orbit Test System," *COMSAT Technical Review*, Vol. 23, No. 1, Spring 1993, pp. 61–99 (this issue).

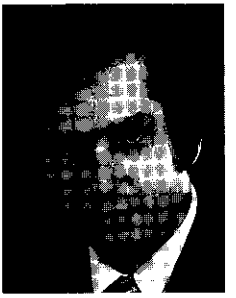
[5] V. Riginos *et al.*, "In-Orbit Test and Monitoring Systems Architecture," AIAA 14th International Communications Satellite Systems Conference, Washington, DC, March 1992, *Proc.*, pp. 951–961.

[6] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*, Hillsdale, NJ: Lawrence Erlbaum, 1983.
[7] *OSF/Motif Style Guide*, Rev. 1.1, Open Software Foundation, Englewood Cliffs, NJ: Prentice-Hall, 1991.
[8] *Apple Human Interface Guidelines: The Apple Desktop Interface*, Apple Computer, Inc., Cupertino, CA, 1987.
[9] G. Booch, *Object Oriented Design With Applications*, Redwood City, CA: Benjamin-Cummings, 1991.
[10] B. Stroustrup, *The C++ Programming Language*, Reading, MA: Addison-Wesley, 1986.

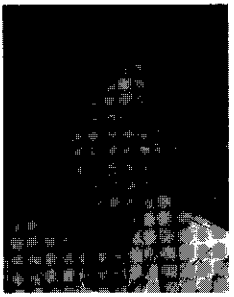
Kenneth D. Fullett received a BSEE and MSEE from the University of Illinois, Urbana-Champaign, in 1979 and 1981, respectively. He joined COMSAT Laboratories in 1981 as a member of the Transponders Department of the Microwave Technology Division and participated in all aspects (including both microwave hardware and computer software system design) of many IOT systems, including those for INTELSAT, MCI, and EUTELSAT. His work also involved software development for COMPACT Software, and he was a project manager for the RF Terminal Supervisory System of the NASA/ACTS earth station. Mr. Fullett is currently engaged in high-energy physics research in the Anti-Proton Source Department of the Accelerator Division of Fermi National Accelerator Laboratory, Batavia, Illinois.



Walter D. Kelley, Jr., earned a BS in electrical engineering at the Catholic University of America, Washington, D.C., in 1974; and an MBA at Marymount University, Arlington, VA, in 1983. In 1991, he joined the Transponders Department of the Satellite and Systems Technologies Division at COMSAT Laboratories as a Member of the Technical Staff. At COMSAT, he has participated in development of the IOT system for Hughes Communications' DirecTv™, an IOT system for EUTELSAT, and the NASA/ACTS ground station control and status subsystem.



Vasilis E. Riginos received a BE, MEng, and PhD in electrophysics from the Stevens Institute of Technology, Hoboken, NJ, in 1964, 1970, and 1973, respectively. He is currently Manager of the Transponders Department of the Satellite and Systems Technologies Division at COMSAT Laboratories, where he is responsible for directing research and development on communications system performance as applied to satellite transponders. He also supervises research and development in advanced microwave circuits such as high-power amplifiers, regenerative receivers, filters, and multiplexers. Dr. Riginos participated in the evaluation of the Inmarsat program, and has been project manager for the GTE ATEF IOT system, the INTELSAT Maritime Communications Subsystem IOT station, the EUTELSAT IOT system, and the Hughes DirecTv™ IOT system. He is a member of Sigma Xi, AAAS, IEEE, and the American Physical Society.



Pei-Hong Shen received a BS and MS in genetics, and an MS in computer science, from Washington State University in 1983, 1984, and 1986, respectively. From 1978 to 1981, she studied biology and genetics while attending Fudan University in Shanghai, Peoples Republic of China. Ms. Shen is currently a Senior Member of the Technical Staff in the Transponders Department of the Satellite and Systems Technologies Division at COMSAT Laboratories, where she is primarily responsible for design and development of software for communications satellite applications. Since joining COMSAT in 1987, she has been involved in the design and development of the following systems: NASA/ACTS, EUTELSAT IOT, and Hughes' DirecTv™ IOT. She is a member of the IEEE Computer Society.

Steven L. Teller received an AS and AA from Harper College in 1979; and a BA in information and computer sciences from Hood College in 1991. He is currently a Member of the Technical Staff in the Transponders Department of the Satellite and Systems Technologies Division at COMSAT Laboratories. Since joining COMSAT, he has been involved in various aspects of the IOT of communications satellites, including the NASA/ACTS RF terminal supervisor, and IOT for EUTELSAT, MCI, INTELSAT, GTE, and the Hughes DirecTv™ system. He has been responsible for software vs manual measurement verification during system development, in-plant testing, and on-site testing. He was also involved in prototyping and testing various new measurement schemes, and was a major contributor to final system installation testing.