DOCUMENT RESUME

ED 082 488 EM 011 457

AUTHOR	Schultz, Gary D.
TITLE	The CHAT System: An OS/360 MVT Time-Sharing Subsystem for Displays and Teletype. Technical Progress Report.
INSTITUTION	North Carolina Univ., Chapel Hill. Dept. of Computer Science.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	UNC-TPR-CAI-6
PUB DATE	May 73
NOTE	225p.; Thesis submitted to the Department of Computer Science, University of North Carolina
EDRS PRICE	MF-$0.65 HC-$9.87
DESCRIPTORS	Computer Programs; Input Output Devices; *Interaction; *Man Machine Systems; Masters Theses; Program Descriptions; *Systems Development; Technical Reports; *Time Sharing
IDENTIFIERS	*Chapel Hill Alphanumeric Terminal; CHAT; CRT Display Stations; OS 360; PL I; Teletype

ABSTRACT

The design and operation of a time-sharing monitor are described. It runs under OS/360 MVT that supports multiple application program interaction with operators of CRT (cathode ray tube) display stations and of a teletype. Key design features discussed include: 1) an interface allowing application programs to be coded in either PL/I or assembler language; 2) use of the teletype for subsystem control and diagnostic purposes; and 3) a novel interregional conduit allowing an application program running under the Chapel Hill Alphanumeric Terminal (CHAT) monitor to interact--like a terminal operator--with a conversational language processor in another region of the OS/360 installation. (Author)

University of North Carolina at Chapel Hill

Department of Computer Science

THE CHAT SYSTEM: AN OS/360 MVT TIME-SHARING SUBSYSTEM FOR DISPLAYS AND TELETYPE

Gary D. Schultz

May 1973

Technical Progress Report CAI-6
to the
National Science Foundation

under Grant GJ-755

DEPARTMENT OF COMPUTER SCIENCE
University of North Carolina at Chapel Hill

THE CHAT SYSTEM: AN OS/360 MVT TIME-SHARING SUBSYSTEM FOR DISPLAYS AND TELETYPE

by

Gary D. Schultz

A thesis submitted to the faculty
of the University of North Carolina
at Chapel Hill in partial fulfillment
of the requirements for the degree of
Master of Science in the Department
of Computer Science.

Chapel Hill

1973

Approved by:

Adviser

Reader

Reader

GARY DENNIS SCHULTZ. The CHAT System: An OS/360 MVT Time-Sharing Subsystem for Displays and Teletype. (Under the direction of DR. FREDERICK P. BROOKS, JR.)

This thesis describes the design and operation of a time-sharing monitor running under OS/360 MVT that supports multiple application program interaction with operators of CRT display stations and of a Teletype. Some key features of the design are (1) an interface allowing application programs to be coded in either PL/I or assembler, (2) use of the Teletype for subsystem control and diagnostic purposes, (3) a novel interregional conduit allowing an application program running under the CHAT monitor to interact--like a terminal operator--with a conversational language processor in another region of the OS/360 installation.

ACKNOWLEDGMENTS

I extend my profound appreciation to William H. Blair for his collaboration in making the CHAT System a working reality. Throughout the two-and-a-half years of system development, I benefited daily from his insights and assistance and his unrivaled dedication to quality work. His companion thesis [B3], and the many references to it herein, only reveal his technical contributions to the project. For his other contributions, I thank him heartily.

I also thank J. Craig Mudge for his willingness to assist in testing sessions at all hours of day and night--sacrificing his own scarce time to do so.

Dr. Frederick P. Brooks, Jr., gave valuable criticism and advice on early drafts of this thesis, improving its organization, style, and clarity. For this, the readers will perhaps thank him as much as I do.

Partial support of the project was funded by the National Science Foundation under NSF Grant Number GJ-755.

TABLE OF CONTENTS

The way to avoid the machine taking command is not to take more and longer vacations from a life dominated by machines, or from a machine regulated existence. The solution lies in finding ways to make this an age where humanity dominates despite the usefulness of machines, and to do this by making fullest use of their convenience....

...The advantages of the machines are so obvious and so desirable, that we tend to become, small step by small step, seduced into ignoring the price we pay for their unthinking use. The emphasis here is on unthinking use, because they all have their good uses. But the most careful thinking and planning is needed to enjoy the good use of any technical contrivance without paying a price for it in human freedom.

Bruno Bettelheim
The Informed Heart

CHAPTER 1: INTRODUCTION

The Chapel Hill Alphanumeric Terminal (CHAT) System is a complex of computer-linked terminals for which access to a library of interactive programs is supported by a time-sharing monitor operating under OS/360 MVT. The terminal complex consists of (1) a cluster of display stations sharing, by program-controlled multiplexing, a single link to the computer and (2) a Teletype connected by common-carrier dial-up facilities. The program library includes a number of application programs created by students and faculty of the University of North Carolina Computer Science Department as part of the Department's research project in computer-assisted instruction (CAI). These programs cover a variety of applications, including a gamut of student-, teacher-, and author-controlled CAI as well as some on-line services similar to those present in other time-sharing systems.

With respect to the CAI Project emphasis and intended terminal usage, the CHAT System centers around the display stations, each of which is a multicomponent unit--including keyboard and lightpen for data entry and four-color CRT and random-access slide projector for display of output. The program library reflects this focus, with most applications of importance and complexity being designed for interaction with the display stations. The Teletype plays an important role in terms of remote operator control of the CHAT programming subsystem--somewhat akin to the computer console control of the overall installation multipro-

gramming system. By means of a primitive command language, the Teletype operator can communicate directly with the CHAT Monitor for purposes of subsystem control and fault diagnosis. It is also possible to invoke an application program from the Teletype and, indeed, subsystem control can be interleaved with application program interactions if desired.

Some of the application programs in the CHAT program library antedate the existence of the CHAT System, having undergone interface and/or device-adaptive changes prior to inclusion in the system. Brownlee's PAMELA [B4], an interactive assembler for the System/360 assembler language, was originally written for a standalone CC-30 display terminal--the same display used in the CHAT cluster--but using a display I/O interface entirely different from that presented by the CHAT Monitor. Pikaplot [B2] was adapted for a standalone CC-30 from Oliver's numerical analysis laboratory simulator [O1] for the IBM 2250 graphics display. Pikaplot exploits the slide projector to capture some of the power of the 2250's graphics facility and originally used a display I/O interface different from both PAMELA's and CHAT's. Hypertext [C1] , Brown University's text editing system, also for the 2250, was adapted by Wait [W1] for the CHAT System displays directly.

Of the major members of the program library, only Mudge's DIAL [M1] was developed exclusively with the CHAT System in mind. DIAL is a total computer-administered program instruction (CAPI) system, including author mode, student mode, its own comprehensive file management, record-keeping, and facilities for checkpoint/restart. DIAL's author mode allows for considerable enlargement of the CHAT library: authors can create an ever-expanding number of authors' programs that

can be executed on the "DIAL machine" in student mode for CAI purposes.

Another member of the library that was developed exclusively for the CHAT System is a clever program by Blair [B3] that exploits an "interregional conduit" facility of the CHAT Monitor. This facility, intended primarily to enhance the power of DIAL, is exploited by Blair's program to allow interaction between the display terminals and the IBM teletypewriter-oriented Conversational Programming System (CPS) [I1], an interactive PL/I-dialect language processor residing in a separate region from the CHAT programming system.

The CHAT System has brought these several applications together into a single subsystem library. The development of a time-sharing subsystem monitor and a common terminal I/O interface supports both multiapplication operation and multiterminal access within a single region of an OS/360 MVT multiprogramming installation.

CONCERNS AND ORGANIZATION OF THE THESIS

The primary concerns of this thesis are the design and facilities of the CHAT Monitor and the dynamics of the CHAT System operation. With the exception of one application program, OLTEST (On-Line Test)--a diagnostic extension of the Monitor--little will be said here concerning the details of the members of the CHAT subsystem application library; these are documented in the previously cited references. Frequently, the application programs are referred to only generically as application subtasks--reflecting their subordinate role in the task-control hierarchy within the CHAT programming region of the OS/360 installation. The CHAT Monitor has no scheduling- or control-bias or any other sensitivity based upon the particular application program(s) in execution.

The basic concerns of the thesis are: (1) the facilities offered by the Monitor, including the interfaces visible to terminal operators and application programs, the management of the terminal and execution-time resources, and the mechanisms provided for CHAT subsystem control; (2) the internal programming structure of the Monitor; and (3) the philosophy and constraints influencing the Monitor design. In the following paragraphs we elaborate somewhat on these concepts and describe their relationships to the organization of the remaining chapters of the thesis.

The Monitor acts as the intermediary for all communication exchanges between terminal operators and application programs. In this role, the Monitor provides an interface to each side that hides the complicated control mechanisms involved. Chapter 3 describes the PL/I application program interface to the CHAT System resources--display equipment,

Teletype, CPS processing services. A semantically identical and syntactically similar (identical except for minor variations due to unavoidable language differences) assembler macro interface for assembler-coded application programs is described by Blair [B3]; common Monitor procedures are invoked from both languages. The interface provides a high-level appearance of the resources to the application program, hiding the complex multistep mechanisms involved in control of the equipment and in System/360 I/O programming.

The interface to the terminal operator, because of its unobtrusiveness, is less explicit. Chapter 2 (see "Hardware Specifics") describes the simple terminal operator keyboard usage required to send data from either a display station or the Teletype to an application program. While the idea was to simplify display station keyboard operation as much as possible (it is function-key ridden, allowing the possibility of overly low-level operator control of equipment operation), the idea for the Teletype was to enhance its capability. Thus, certain keys on the Teletype are given special meaning for line- and character-editing purposes.

Taken together, Chapters 2 and 3 show a certain intended symmetry in the design of the terminal and application interfaces. The application interface resembles operator keyboard conventions in its provisions for (display) cursor control, message formatting, and lack of concern for primitive non-data functions. Both interfaces feature Monitor-reporting of pathological circumstances, such as application program failures (via "proctor messages" to the terminals) and equipment failures (via "on-unit" condition-signalling to the application programs). Each side is given aid related to time-lapse abnormalities at the other side

of which the Monitor is not aware. For the application interface, provision is made for allowing interval time-outs of program read-operations, whereupon the application program is reactivated and free to try a new operation. At the terminal-side, abnormal time lapses are humanly evident; here the Monitor allows certain operator conventions such as "extra-interrupts" and operator-controlled aborting of the application program to test for or to eliminate unresponsive, looping programs. Finally, each interface provides a view of one member at the other end: just as an operator deals only with one application program (at a time), so each application program is coded to "see" just a single operator--even though a single application (via Monitor usage of OS/360 multitasking) may be serving many operators.

In the multiterminal/multiapplication environment of the CHAT System, resource multiplexing is of key importance in determining the responsiveness of the CHAT System from the viewpoint of the terminal operators. The two resources that require multiplexing, or use-sharing, are the single link connecting the display cluster to the computer and execution-time on the central processing unit (CPU). Chapter 2 and Chapter 5 describe the display multiplexing, while Chapter 4 describes the Monitor's sharing of CPU-time among the application subtasks. The latter multiplexing involves two types of scheduling by the Monitor: en bloc application scheduling in accordance with a formula edicted by the multiprogramming installation manager (for CHAT- vs. other-region use of the CPU) and sharing among CHAT application subtasks of the time-slice allotted to CHAT by the first scheduling formula.

Subsystem control refers to console-like control of the CHAT subsystem for such capabilities as region-shutdown, subtask aborting, and

fault diagnosis. The Teletype serves as the subsystem console for these purposes. Chapter 6 describes the use of the Teletype for this role, while Chapter 8 describes an application program invokable from the Teletype which greatly enhances the on-line testing capability for equipment diagnosis.

The internal programming structure of the Monitor is outlined in Chapter 2 and detailed in Chapters 4-8, where the separate components are discussed individually. Chapter 2 gives an overview of the dynamics of subsystem operation--how the components of the Monitor communicate and work together and how application subtasks are linked to Monitor-controlled resources. The later chapters deal more fully with the functions of the separate components acting in isolation from each other.

Insofar as design constraints and design philosophy are concerned, it is difficult to know, much less to expose, all facets of these influences on the design of a system. Frequently, a designer's initial theoretical sense of what constrains a design (restrictively) or enhances it, changes after further experience and practice. Some constraints can be imagined, while certain biases are not even recognized--simply due to muddle-headedness or lack of perspective on the part of the designer.

Such faults are certainly present in the design and exposition of the CHAT Monitor. (But then Computer Science, itself, is more engineering than science!) The attempt throughout the thesis is to expose what the author recognized as motivating the design and as impinging on design alternatives. Here, we cite only two major factors affecting the design.

The major constraint on the CHAT Monitor design, apart from the equipment itself, was the necessity to run within a single region of a multiprogramming installation controlled by OS/360 MVT. This meant that

CHAT requirements had to be compatible with the broader installation perspective on system usage, but at the same time gave CHAT the powerful multitasking capabilities provided by the MVT-version of OS/360 to achieve its design objectives. Chapter 2 weighs some of the advantages and shortcomings of this dual-faceted constraint, while later chapters frequently reiterate its consequences.

A second major factor, more a bias than a constraint, was that the CHAT System operation and design must exhibit the quality and robustness of a production system. No formal or rigorous measure of the degree to which CHAT meets a production standard is possible, since the standard itself is wholly subjective. Some of the effects of this standard on the CHAT design include (1) insulation of the CHAT Monitor from misuse or abuse of the terminal and application interfaces and protection of one application program from another, (2) exhaustive analysis and retry of all transient and "hard" failures in equipment operation, (3) logging and reporting (where useful) of abnormal conditions, (4) diagnostic services geared to abnormal states of the equipment, (5) long-life considerations: design features avoiding control program release-dependencies and promoting ease of extension for obviously anticipated growth of installation equipment, and (6) efficient use of installation resources, most especially main storage. Various other implications could be cited but these should suffice to show the fundamental application of the idea. The author's choice of antonym for production is experimental (or short-term).

With this introduction to the concerns and organization of the thesis, four reading strategies are suggested for the remainder of the thesis. The application programmer interested in writing a program to run

on CHAT should read Chapters 2 and 3. The CHAT System proctor should read Chapters 2, 6, and 8 to learn all aspects of terminal and command usage (and also to find out what proctor means). The interested reader should read the entire thesis. The systems programmer inheriting responsibility for major extensions, revisions, or maintenance of the CHAT Monitor should be an interested reader but, in addition, should thoroughly pore over the code listings issued under separate cover as an appendix to the thesis. For all reading strategies, where Blair's thesis [B3] is referenced and is necessary to full understanding, it should be consulted.

As a final recommendation, when the use of acronyms becomes too dense, consult Appendix A for the meaning of those less commonly used and most easily forgotten.

RELATED LITERATURE

A number of references are recommended to readers of this thesis--either because they directly supplement or illuminate the description or because they discuss similar or alternative versions of CHAT design features.

For background purposes, references on OS/360 are useful: the terminology of this thesis derives from IBM usage and frequent mention is made of OS/360 interfaces and services. Witt's article [W4] and the IBM concepts manual [I3] are excellent introductions. IBM publishes a large number of manuals on OS/360--too many to cite here; [I4] and [I5] are sufficient to learn more about the specific OS/360 supervisory services and macro instructions mentioned in the thesis, while [I6] and [I7] provide more than enough exposure to other aspects of OS/360 to satisfy the readers.

An application programmer reading Chapter 3 will probably already be acquainted with the IBM reference manual on the PL/I language [I10]. Similarly, to use the programming interface for access to CPS or to better understand the interregional conduit, the reader should know the contents of the IBM [I1] or TUCC [T1] references.

Only the systems programmer inheriting responsibility for the CHAT Monitor will be interested in the manufacturer manuals on the display equipment [C2]-[C6]. The material ("Hardware Specifics") in Chapter 2 and hands-on experience at a display station should satisfy the needs of any other reader.

Certain articles are of great "cultural" value, although not directly related to the CHAT programming environment. Dennis and Van Horn

[D2] provide alternative terminology (widely used outside of IBM) for concepts similar to those in OS/360--especially with regard to multitasking. Indeed some of their generalizations of the task ("process") concept surpass those implemented in OS/360 MVT--a matter further discussed in Chapter 2.

Denning's fine article [D1] on modern control program design is a rich source of tutorial and survey material on the history, trends, and basic concerns of existing systems. It is useful reading in order to place OS/360 and, indeed, CHAT itself, in proper perspective with regard to other systems. One particular idea of Denning's (and others) is to characterize the combined hardware/software interface visible to a program as a machine.

An idea due to Dijkstra [D3] involving layering, or onion-skin design, provides an interesting modification of this concept. Dijkstra describes a design of a control program whereupon the basic hardware of a computing system is completely enclosed by the innermost or most primitive layer of the control program. This layer would in turn present for all capabilities below it a completely new, possibly extended or even different, interface to the next layer in the onion above it. Hence each successive layer presents, in similar fashion, a total, new machine. The concept is shown by Dijkstra to give added modularity of design, testing, and control and to promote debugging and proof of program correctness.

Neither CHAT nor OS/360, itself, exhibits such layering. OS/360 and System/360 together represent a machine in Denning's sense, but OS/360 is not a total layer for the System/360. The CHAT Monitor also is not a layer in Dijkstra's sense although the combination System/360, OS/360,

and the CHAT Monitor comprises a machine. The CHAT Monitor does form a layer over the CHAT System equipment and does provide some layered control (subtask management)--invisible to applications--but application programs have completely free and unmonitored access directly to OS/360 and System/360 facilities. Two disadvantages of layering are obvious: each layer requires increasing programming effort and the time to cross layers may inhibit performance. (One measure of layering performance is that functions in a lower, more primitive, layer should be accomplishable in a time negligible with respect to the time scale in the next layer above [L1].) The first disadvantage determined CHAT's layer-lessness.

In later chapters of this thesis, comparisons relating to program design and control program usage are made among CHAT and various IBM-released interactive programming systems built on OS/360. CPS [I1] is an interpretive system offering interactive support for teletypewriters and a language interface which is a dialect of PL/I. While CPS, like other interpretive systems, does represent a closed machine interface, it does not fully layer all capabilities below it--some facilities are simply not available to its interface users. The Real-Time Operating System (RTOS) [W2] and the Time-Sharing Option (TSO) [S1] are nonlayered OS/360 extensions built on, respectively, pre-release and late versions of MVT. These are discussed further in Chapter 2.

Another aspect of the CHAT Monitor worth comparing with other system designs is the means provided for terminal- and CPS-access. In the case of terminal-access, CHAT was faced with two design problems: (1) how to program its own terminal control logic (at the assembler language level) using OS/360 and (2) what form of high-level interface to present application programs, which were intended to be coded pri-

marily in PL/I.

The Monitor support for the CCI display equipment had to be built on the OS/360 Execute-Channel-Program (EXCP) interface [I4] [I7] since CCI offered no support package and the IBM support for its own manufactured displays was not usable because of fundamental differences in device operation. In the case of the Teletype, the IBM access method BTAM [I2] does provide support but this was considered inadequate and is not used by the CHAT Monitor--EXCP is.

The form of the high-level interface for application programs was designed without knowledge of other systems. No display-access support exists in the IBM PL/I language definition and although CCI displays have been widely sold, no literature by CCI customers has been located.

Gwynn [G2] has suggested that CRT terminal access be given more attention in high-level language design and has briefly reported on the interfaces developed at his own institution, Stanford University, and at California Institute of Technology for COBOL and FORTRAN, respectively. The COBOL support is for alphanumeric CRT displays which are polled, while the FORTRAN support is for interrupt-driven graphics terminals. The interfaces described by Gwynn appear to be similar syntactically and semantically to that offered by CHAT for PL/I. In each case, the interfaces use the call-mechanisms of the languages to invoke the terminal-access support.

Anderson and Farber [A1] report an interesting system developed at The Rand Corporation in which their POGO graphics-control support is combined with a Rand variant of the IBM CPS system [I1]. They extend the CPS (a PL/1 dialect) language interface definition rather than use the call-mechanism to give graphics-terminal access to the CPS pro-

grammer. They extend the CPS language by adding: (1) a statement for entering a "construction-mode" whereby POGO facilities are made available, (2) more options in the GET and PUT statements, and (3), an additional ON-condition sensitive to lightpen or stylus actions.

This idea of extending a language definition involves changes to the interpreter and implies continued maintenance of the language from release to release. For CHAT, this approach would entail extension and continued maintenance of the PL/I compiler in use--an undesirable requirement. The idea, however, of building application support on a facility such as CPS, an already existing time-sharing system, has merit--if the language facilities are adequate for the applications' broader requirements.

The CHAT Monitor feature allowing application program access to the CPS program in another region of the multiprogramming installation can also be compared with other schemes described in the literature. Grant [G1], although he did not build his proposed mechanisms, describes in his Ph.D. dissertation the general uses for and design of a "psuedo-teletype" interface to join communicating programs. The interface he proposes is similar to the CHAT interface described in Chapter 3 ("CPS Access"), but he does not provide explicitly for such CHAT-implemented mechanisms as time-lapse signals on read operations, signalling and clean-up operations for one-program-down conditions, or the full capability of the CHAT attention ("ATNCPS") facility. The internal mechanism for controlling the communication--akin to the CHAT interregional conduit described in Chapter 7--Grant terms a stream modulator. One of the applications he considers "grandiose" is precisely what the CHAT conduit allows--namely for a CHAT application program to enter, have compiled,

and execute programs on CPS and then get back output from such programs.

Balzer [B1] describes a generalization he calls ports, an outgrowth of his work on actual mechanisms constructed at the Rand Corporation to run on OS/360 and on the Rand ISPL interactive system (akin to CPS). He defines an interface again resembling CHAT's but omits (at least in his discussion) the capabilities that we mentioned Grant omitted. In his OS/360 version, Balzer used the OS/360 I/O facilities, simply changing a routine address in a system control block referred to in the I/O instruction. This caused a procedure he called a joiner to be invoked at the time of I/O macro execution, rather than the system I/O control. In ISPL, Balzer generalizes the interface to give a program a common port-representation for I/O devices, files, communicating programs, and the supervisor itself. His internal mechanism employs a slight variation of Dijkstra's so-called P- and V-semaphores [D3] for control purposes.

CHAPTER 2: SYSTEM OVERVIEW

This chapter deals with the CHAT System--both hardware and software aspects--as a whole. Besides describing the hardware configuration and programming organization and defining the general dispersion of function among components and modules, the discussion includes a detailed elaboration of the dynamics, interfaces, and communication protocols involved in the operation of the system. These latter concerns are an inevitable source of complexity for any interactive system involving stochastic events, multilevel control, and asynchronous parallel operation. They are stressed in the overview to give the reader a clearer background sense of overall system activity when confronted in subsequent chapters with more detailed accounts of specific functions and modules.

The intent is not simply to describe the design features, but also to reflect on the reasons for and constraints on various design decisions. Where appropriate, alternative methods and other existing systems are discussed.

HARDWARE SPECIFICS

This section describes the CHAT System hardware configuration, the operational characteristics of the terminal equipment, and the manner in which the terminals are used by an operator.

Configuration: Figure 2.1 shows the CHAT System hardware configuration. The central processing unit is an IBM System/360 Model 75 with 256K bytes of fast core and 1024K bytes of large capacity storage (LCS). The terminal equipment includes a Teletype Model 33 with paper-tape perforator and reader, connected by common-carrier dial-up facilities to an IBM 2701 Data Adapter Unit, and a display complex consisting of the following components: (1) a CC-7012 channel adapter, (2) a CC-72 multiplexer, and (3) a cluster of CC-30 display stations.

Each CC-30 display station is a multicomponent unit having a CC-301 controller with 1024 bytes of core storage that performs local device control (character generation, display refresh, buffer storage, cursor control, etc.). A four-color (red, green, yellow, blue) CRT, a light pen, and an alphanumeric keyboard are connected to each CC-301. In addition, each CC-301 has an output channel to which a Kodak Carousel RA-950 random-access slide projector is attached. Except for the slide equipment, all display components are produced by Computer Communications Incorporated (CCI).

The CC-7012 and CC-72 are connected by a 50 kilobit/second "long line" cable. An earlier version of the CHAT System had the CC-7012 and CC-72 connected by means of common-carrier leased-line facilities (including AT&T 201B1 data sets) operating at 2400 bits per second. The

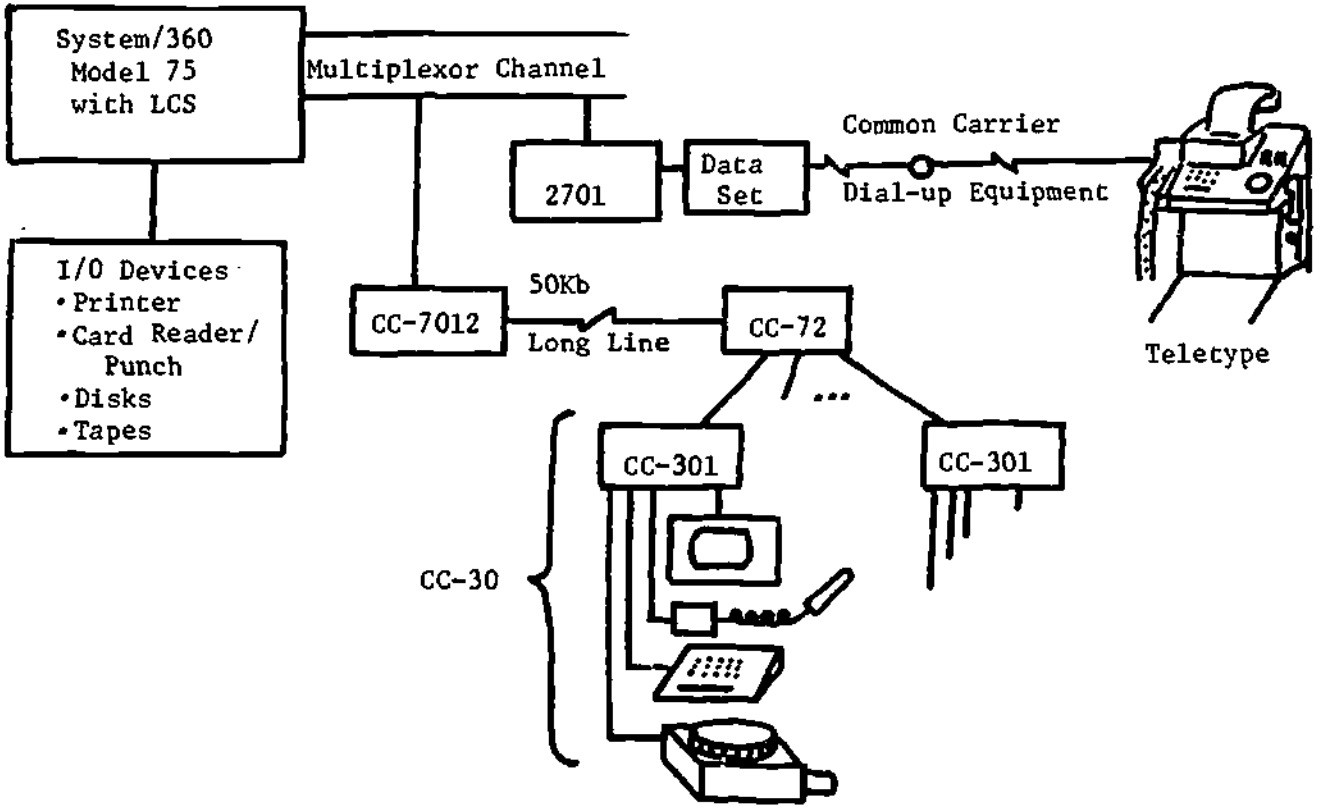


Figure 2.1 CHAT System Configuration

CPU was then located at the Triangle Universities Computation Center (TUCC), a location some twelve miles from the multiplexer. Relocation of the CPU to the UNC Computation Center on the UNC campus permitted reconfiguration to the simpler and higher performance cabling. Significantly, apart from a minor programming accommodation for the increase in data transmission speed, the programming support was insensitive to this change in linkage.

During development of the CHAT Monitor there were five display stations connected to the CC-72 multiplexer; currently, this number is being increased to ten. The multiplexer has a maximum capacity for thirty-two such stations. The CHAT Monitor is coded such that minimal reparameterization and simple reassembly suffice to accommodate an increase in the number of attached displays.

Operation: The display complex is novel and complicated in operation and less familiar to the general reader than the widely used Teletype. Our concern is not with the myriad details that fill several manufacturer's manuals [C2]-[C6] but with the general aspects that have influenced the Monitor design. In particular, we emphasize here the link multiplexing, with details of display station device control deferred to later sections.

The feature of foremost significance is the sharing of a single path to the computer by the various display stations. The operation of the link is similar to that of a telecommunications facility, although here no common-carrier equipment is (now) involved. The link is used for both data transmission and device control, where orders to and status returned from the equipment outboard from the CC-7012 channel

adapter are encoded as characters either isolated from, framing, or embedded in the regular data stream. The link operates serial-by-bit and clock-synchronous with a half-duplex protocol--the latter meaning that two-way traffic proceeds by alternation rather than simultaneously.

Link-sharing and various specialized multipoint attachments are by no means novel in computer-communications systems. However, it is a good deal more common in time-sharing or interactive systems for each remote station to be attached, frequently by dial-up facilities, to a separate multiplexor channel port dedicated to that terminal alone. Where link-sharing is used, it typically involves a terminal polling discipline less elegant than that employed in the CCI complex described here and also less suitable for the particular CHAT environment of low-density, high-speed traffic.

Where individual stations are all attached through dedicated ports, the programming problem is essentially entirely in multiplexing, or time-sharing, the CPU to effect the appearance of response immediacy so crucial to an interactive system. Each station is easily identified through its individual port address in which all higher-level control blocks are rooted. Concurrency and independence of I/O for the separate stations are essentially total.

In the CHAT System configuration there is only one port address for the display complex and, particularly at the line speed of the original configuration on which the Monitor was developed, the I/O for the separate display stations may be interfering. Thus, two new responsibilities, not found in the dedicated-port systems, are imposed on the CHAT Monitor. The first is to provide added support to correctly associate an application with its interacting station. On output this involves additional

addressing so that appropriate routing can occur at the remote multiplexer; on input, additional control exchanges are required to identify interrupting ready stations--an identification no longer implied by the local port address.

The second responsibility concerns the multiplexing of link usage for fair and efficient sharing among the various active display stations. This is quite as complex as the algorithm for multiplexing the CPU resource--involving queueing of delayed requests, scheduling according to priorities related to both transmission direction and operation type, and providing various facilities for continual two-way monitoring of the data link. In addition, the various "boxes" (channel adapter, multiplexer, controller, and device) in the path between source and sink are all quite visible to the Monitor and must be primed and controlled through a mixture of commands and orders differentiated by box--where commands refer to channel program control of the S/360 channel and CC-7012 and orders, to control characters sent to the remote boxes.

The CCI equipment is remarkably flexible in the latitude allowed to the programmer in defining a data link control protocol for reliable and orderly exchange of data, orders, and status over the transmission line. This is a consequence of the fact that the exchange protocol is almost entirely order-driven by the computer program. Hard-wired automatic "responses" such as are typically found in "standard" data link control protocols are absent from this design. For this reason it is almost assuredly true that no two programmers would devise identical procedures for controlling the data link. Nevertheless, the CCI equipment is clearly optimized for the programmer to make use of the attention mechanism for terminal soliciting of input servicing; although, at some expense of the CPU resource, the equipment is capable of being

programmed as a pure polling system. The CHAT Monitor takes full advantage of the attention mechanism but, for priority reasons, does some polling as well.

"State" control and register accessing are both important in programming the attention mechanism. The CC-7012 channel adapter has two states: attention-enabled and attention-disabled. In the enabled state, the CC-7012 will react to the arrival of a special character (ENQ--the "inquiry" character) from the remote multiplexer by signalling the System/360 channel status word (CSW) attention status [I9]; this is done despite the absence of a channel program command pending on the multiplexor channel. To activate this state, or the disabled state, a previously issued channel command must have been executed by the adapter. (Actually, there is another "enabled" adapter-state that is always activated by the Monitor in conjunction with the ENQ-enabled state: this is the enabled-for-any-character state. The need for this stems from real-world vagaries: an ENQ can be scrambled by stochastic noise on the physical link and thus could arrive in unrecognizable condition. Additional "sense" information distinguishes the two events. This detail illustrates that some operational characteristics of any real system are outside the scope of an overview.)

Because the link is half-duplex in operation, there must be no danger of the remote multiplexer sending an inquiry character when the computer is using the link for an output operation, for this would scramble both signals and neither would be recognized. Hence, the program can disable or enable the multiplexer, with respect to its sending ENQ, by transmitting an appropriate order to set its state.

The CC-72 has three registers that can be read or reset under

computer program control: a 6-bit short status (SS) register, a 32-bit station acknowledgment status (SAS) register, and a station interrupt status register having two ranks (SIS-1, SIS-2), each of which is 32 bits. The SS register has only two bits of interest to the program: one bit indicating a bit is "on" in the SAS register, another bit indicating a bit is on in SIS-2. Other bits in the SS register are of interest to a technician using an electronic strobe for equipment diagnosis. The SAS register has a bit to represent each of the maximum (32) number of stations that can be attached to the multiplexer. Every message sent over the link is character- and block-parity checked. If no parity error is detected, a station receiving the message causes its bit in the SAS to be turned on. (This means the SAS must be cleared prior to sending the message since the SAS is unchanged from its previous status on receipt of a bad-parity message.) Reading the SAS register is costly (it is encoded as six characters for transmission) and unnecessary if multiple-station messages are not transmitted without intervening line turnarounds (which they are <u>not</u> under the CHAT Monitor support). Instead, the SS register (encoded as a single character) may alone be read to determine if the previously cleared SAS register is non-zero after message transmission.

The SIS-1 and SIS-2 register-ranks participate in the attention mechanism. Here again each bit in the rank represents a station. When a station operator strikes the interrupt (INT) key on the keyboard, a bit is set in SIS-2. Whenever the multiplexer is enabled, a non-zero SIS-2 causes it to send the ENQ over the transmission line, after which it disables itself. The computer program, provided the adapter is enabled, receives the attention status signal, and transmits an order for

the multiplexer to send the result (a 6-character encoded sequence) of (inclusive) OR-ing SIS-1 and SIS-2. This order has the side effects of also storing the result in SIS-1 (for possible retransmission) and clearing SIS-2. If the computer receives the result error-free, all stations now ready to be serviced can be identified and the SIS-1 rank can be cleared by another order.

Besides the foregoing basic attention mechanism, there are a few nuances of equipment operation. Earlier it was stated that the SS register serves to provide acknowledgment status for message receipt at a display station; a similar need exists for orders sent to the multiplexer. In particular, it is obvious that the computer program requires assurance that an order to enable the multiplexer has been executed. This is particularly compelling because there is no way for the multiplexer to be enabled by an action at the remote site--an obvious hardware design flaw. To meet this requirement, use is made of a convenient capability of the multiplexer: the ability to receive orders together into a single parity-checked character. The Monitor simply simultaneously orders enabling with reading of the SS register. Receipt of the SS is prima facie evidence that the multiplexer is enabled--examination of the SS contents is unnecessary (and irrelevant).

In general, various race conditions and frequent transient errors are indigenous to the CHAT environment; it is, after all, an interactive system with numerous components. And the technology, particularly in the earlier configuration employing common-carrier linkage, is error prone. A good example of a race condition concerns, again, the attention mechanism: an order to disable the multiplexer can be sent at the same time the multiplexer, yet enabled, is sending an inquiry signal.

This contention state is a logical, though rare, consequence of the protocol design and presents no great difficulty. The multiplexer, as noted earlier, disables itself after sending ENQ, holding its registers intact. This provides the necessary hardware cooperation to prevent continued contention. At the computer end, the transmit operation will likely end with unsatisfactory response status indicated, thereby invoking Monitor error recovery procedures to retry the operation. Notice that the requirement for the SIS-2 rank is a consequence of another potential race condition exacerbated by the necessary delay between reading and clearing of the SIS-1 rank. Other race conditions concerning interaction conflicts, not strictly involving hardware, are more visible to the Monitor which invokes programmed precedence decisions.

Transient errors generally result from the vulnerability of the data path to environmental electrical noise. Again, these are inevitable and, as is the case for the race conditions, their accommodation is basic to the design of the Monitor. Because of the sizable number of different operations, the complex combination of primitive steps comprising each operation, and the different errors that can occur, the error recovery procedures, which are intended to be exhaustive, form a considerable bulk of the Monitor code. They are, however, fully justified by the desire for robust operation of the system--one designed for production use.

Finally, there is an intrinsic disadvantage of the hardware configuration. This is the availability exposure attending use of the single link. Any "hard" error that occurs between multiplexer and channel adapter, inclusive, results in total display-system outage. Given the objective that the CHAT System would be of production system quality,

some diagnostic capability was clearly needed for the CHAT equipment. A two-fold complication constrained the designer's options for providing this capability: (1) manufacturer-provided diagnostic programs were incompatible with the installation operating system and required a dedicated System/360 to run; and (2) the multiprogramming installation could not generally be dedicated to CHAT System requirements because of other user needs.

These circumstances levied a basic requirement that the CHAT Monitor provide on-line diagnostic tools. The substance of this support, which uses the Teletype as a control console to drive the programmed diagnostics, is discussed in detail in later sections.

Terminal Usage: In this section we are concerned primarily with the mechanics of terminal usage. The information content of transactions depends on the particular application program invoked. The Monitor serves only as the controlling intermediary between terminal and process.

In the case of the display station, the idea was to keep the mechanics as simple as possible for the station operator. The display keyboard is equipped with a number of function keys, along with the necessary character generating and cursor control keys. A number of these function keys are required only for off-line operation, being redundant in on-line operation where Monitor program control can supplant their use. Their presence is another indication of the flexibility of the CCI equipment for accommodating different control strategies. Various tradeoffs are possible, allowing the mechanics of data transmission to be shared in different fashion by terminal operator and control program, depending on the varying emphases placed on human factors and pro-

gram complexity. For example, message transmission from the display to the computer involves various steps for cursor placement, end-of-text marking, and transmit-initiation which can be controlled at the display keyboard but which the CHAT Monitor performs through its own control sequences. The intent of the CHAT Monitor support is to do as much in programming as is possible.

In first encountering an idle CHAT display, the operator need only depress the interrupt (INT) key. This leads to the responding display on the CRT of a list of CHAT application programs and initiates an interaction session. This "table of contents" of available programs is shown, sans color highlighting, in Figure 2.2. The operator can lightpen one of the listed program names or else type in one of those or an unlisted one. In either case, the program must reside in the CHAT library, which is defined through the JOBLIB facility of the OS/360 Job Control Language (JCL) used in initiating the CHAT job. Usually an operator would type only to specify the name of a program recently added to the CHAT library but not yet listed in the displayed contents. The program listed as TASKLIB is a special program, written by Blair [B3], that allows on-line specification of a program contained in a private library--a useful aid during new program development. Further details on TASKLIB and the library facilities are given in [B3].

Transmission of keyed-in data is very simple: after keying in the data, the operator merely depresses the INT key. This automatically locks the keyboard and, by the previously discussed attention mechanism, alerts the CHAT Monitor. The actual reading operation is a multistep process initiated by the Monitor. It involves initialization of the multiplexer and the display controller, reading the cursor coordinates

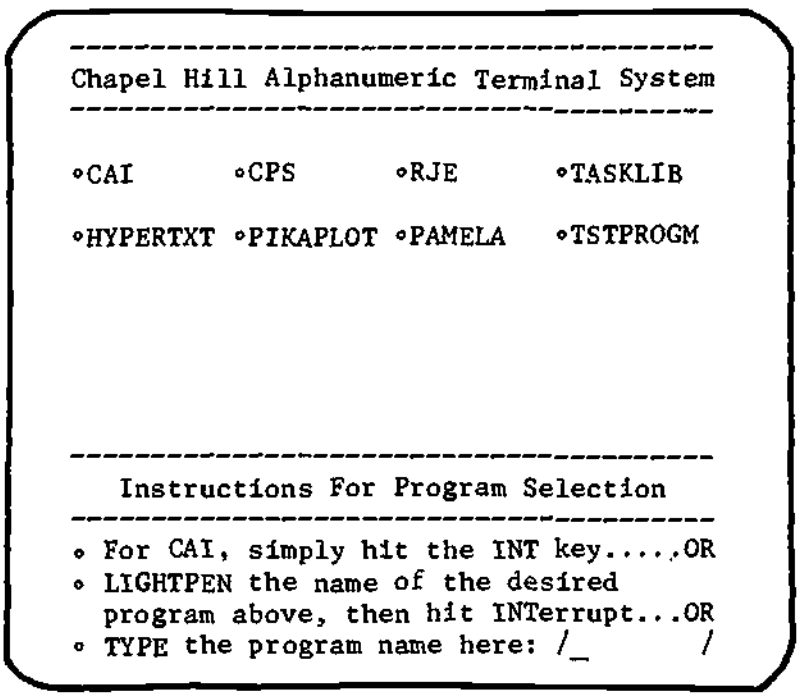


Figure 2.2. CHAT Application Program "Table of Contents"

(contained in a controller register), saving the character over the cursor, placing a special control character (ETX--end of text) in its place to stop the read, setting the cursor to a program-determined starting location preceding the final cursor location, ordering the controller to transmit data between the cursor start location and the ETX, and then restoring the previously stored cursor character and the cursor itself as they were at the time of interrupt. Of course, some status exchange also occurs to check the success of particular steps. At the end of the operation, the keyboard is still locked. It is unlocked in response to an application program command or issuance of a new read request.

A light pen operation is similar. The operator illumines the background of some character position on the CRT and again initiates an interrupt. This interrupt is signalled using the interrupt button on the light pen barrel. Completion of the subsequent read operation, also a multistep process, causes the search character illumining the light penned position to be removed from view.

Both operations--lightpen reading and message reading--are visible to the display operator, although on the current 50-kilobit/second linkage a message-read takes a long message and a quick eye to see the cursor motion. The Monitor gives high priority to a read operation, higher than that given, for example, to a program-initiated write operation. This allows the operator quick assurance of completion of the "mechanical" aspect of his input operation where delay is less tolerable than it is in awaiting the responding output from the application program. The interference between display stations is obviously less for the current linkage than it was for the 2400 bit/second linkage on which the Monitor was developed.

While the Monitor is generally invisible to the display operator, it can participate in visible fashion when the application program in use runs into difficulty. For example, the application program can end abnormally. In this case, the Monitor sends a proctor message to the affected display station. This message includes a code denoting the type of termination involved. A code prefixed by "S" indicates that the operating system initiated program termination; the three hexadecimal digits in the code are system-defined and their meaning is given in the IBM manual on system completion codes. A code prefixed by "U" followed by four decimal digits defines a termination initiated by application program issuance of an operating system ABEND macro instruction. The code is application-program defined. A four-digit decimal code without a letter prefix denotes an application program termination involving a non-zero "return code." The code displayed at the CRT represents the return code and is also application program dependent. A contrived termination involving a proctor message is illustrated in Figure 2.3.

An application program can run into difficulty without actually terminating. Here, another Monitor facility may be useful. If the application program suddenly appears "dead" (perhaps because of a program loop), the operator may enable the keyboard, if necessary, by hitting the "master clear" key; type in "ABORT" (or "abort") anywhere except split across the bottom and top lines of the screen; and hit the INT key, master clear key, and INT key in succession. If a previous interrupt was not acted upon, only the first interrupt in the above sequence is necessary. The Monitor recognizes the extra "unsolicited" interrupt, reads the abort request, and terminates the application program. (Extra interrupts sent without an abort request cause the Monitor to place a "?"

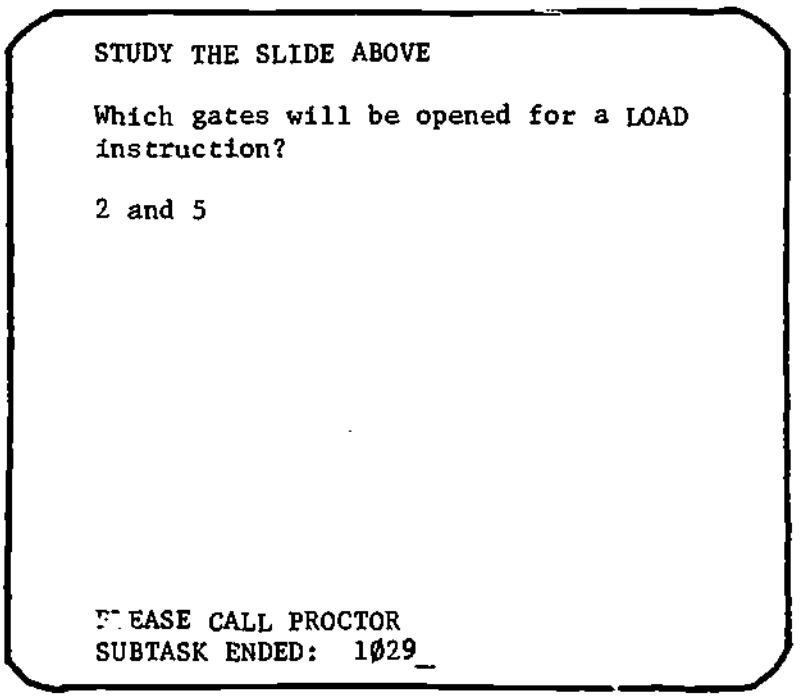


Figure 2.3. Example of Application Termination with a Proctor Message

character at the cursor location.) This is useful if the difficulty involves a program delay, but will not otherwise always work. For this reason, the application program can also be terminated by entering a $ABORTnn command at the Teletype, where "nn" denotes the display station from which the program was invoked. All of these facilities are particularly valuable in the course of on-line testing and development of new application programs.

Operetor use of the Teletype is also simple. Connection to the CPU is established by a dialing operation. In place of a table of contents display the operator is given a welcome message from the Monitor. A set of commands, described in detail in a later chapter, is available for direct interaction with the Monitor. An application program can be invoked by typing in the $XEQ command followed by the program name. The commands to the Monitor can be used even while an application program is active. This is the reason for the $-prefix on the Monitor commands: it distinguishes them from all application program transmissions.

Operator messages to the computer are always ended by the X-Off control character. Line-and-character-editing are provided by the Monitor. Cancellation of a line is indicated by using the X-On character rather than X-Off to signal message completion. Character deletion can be done by use of the underscore or back-arrow character, whichever is present on the Teletype in use. The Monitor signals its acceptance of a Teletype message by sending carrier-return and line-feed to the Teletype. When it is ready to resume reading it prints a "?" character at the beginning of the fresh line.

Special use is made of the "Break" button. This allows the operator additional capability to interleave Monitor commands with applica-

tion program interactions. The Break function is useful when the operator wants to get the attention of the Monitor while the application program is not ready to read (no "?" character has yet been printed) or during the period when the application program is printing many lines of output. The Break signal causes the Monitor to respond immediately with an invitation to the operator to enter a command. Once the command has been read and its execution reported in the form of a Teletype output message by the Monitor, the application program activity is resumed. In the case of an interrupted output sequence, the Monitor retransmits the interrupted line so no data is lost. One obvious use for this Break facility is in the case, again, of a faulty application program. Here, the operator may wish to enter the $ABORT command to terminate the application program.

Just as for the display stations, the Monitor provides failure reports on Teletype-invoked application program. In these cases, the "Please call proctor" part is omitted, because the proctor is generally the sole user of the Teletype. Additional details on the command language, of interest more to a proctor than to the general reader, are given in Chapter 6.

ORGANIZATION AND FUNCTIONS OF THE MONITOR

The decision to build the programming support for the CHAT System within the available System/360 multiprogramming system was never a matter for debate. The cost-effectiveness of including the CHAT support, which clearly does not require a dedicated machine, as a single LCS-based service sharing the multiple-service installation with other users was justification enough for this decision. Also, the extensive, and familiar, program library facilities and the high-level "machine" presented by OS/360 were equally important considerations. Thus, application programs could be written in PL/I and could exploit the high-level OS/360 data management services for disk-based files--even sharing access to them with other programs entered through the independent batch facilities of the installation.

The facilities of the operating system of central concern to the design of the Monitor are those for multitasking. These facilities, and the task concept itself, are one of the leitmotifs of this thesis, warranting some preliminary background discussion.

The key abstraction developed for OS/360 MVT was the notion of the task, a concept apparently originated concurrently in other design projects of the early 1960's (cf. Denning [D1])--in particular MULTICS (cf. Dennis and Van Horn [D2])--and denoted by the name process (perhaps less appropriate a term because of the ambiguity of "multiprocessing"). Essentially, the task concept resulted in the downward extension of the facilities for scheduling, setting priorities, accounting, and partitioning which had been necessary for multiprogramming among independent jobs, to the cooperating modules of a single job. Additional problem program

interfaces to these control program facilities were added to permit such cooperation. It was, perhaps, a result of insufficient foresight that the OS/360 implementation is characterized by multiprogramming without adequate provision for interregional or job cooperation (running programs should also have been recognized as system resources)--a matter discussed further in Chapter 7--and by cooperative subtasking without sufficient protection, which is discussed more fully in a later section of this chapter. Both issues reflect an operating system (and System/360) design dominated by a batch operation orientation rather than one for interactive time-sharing; neither issue is crippling in practice.

Two other concepts of OS/360 are appendages and exit routines, both of which are used in the CHAT Monitor. Appendages are user-written program extensions to control program supervisory facilities which are invoked as part of the supervisory control of such events as I/O interrupts. Exit routines are used by the Monitor in its handling of timer-events in its time-slicing implementation (Chapter 4). They involve a type of autonomy of execution for the exit routine of a task that resembles Conway's coroutine notion [C7]. An exit routine to handle timer events can be specified by a task (by use of STIMER) to be scheduled and the subsequent activation of the exit routine preempts but does not otherwise change the execution-flow of the task.

For efficiency reasons, appendages and exit routines may also be used to complement the tasking facility. For example, the IBM special-purpose "hypervisory" real-time monitor (RTM) [I8] uses appendages to avoid costly task-switching on real-time responses. The real-time operating system (RTOS) [W2], designed by IBM and NASA for the Apollo program, devised an alternative idea of a "system task" requiring only four per

cent of the overhead of the conventional OS/360 attaching mechanism.

In the following subsections, we indicate how these OS/360 facilities are employed by the CHAT Monitor for its control purposes.

Task Structure and Control: Basically, the CHAT Monitor consists of three control tasks. In order of dispatching priority, these are: the Monitor Subtask Scheduler (MSS), the Monitor I/O Scheduler for displays (MIOS), and the Monitor I/O Scheduler for the Teletype (MTWX). The term monitor is also more generally applied in this thesis to include components not part of the three named tasks. This includes code such as the subroutines used in the interface between the application programs and MIOS or MTWX. These are packaged with the other Monitor code but operate under control of the application task. Another component facility, the interregional conduit, includes code that runs, in part, under the same task control as MIOS and, in part, as an extension of the control program. Indeed, it is further complicated because the code is located outside the CHAT region (in "LINKPACK"); in addition, some CPS appendage code (in the CPS region) executes under MIOS task control!

The division of the Monitor into three control tasks is primarily modular by function. This was important initially in the design and testing stages of the Monitor development. It allowed a cleaner separation of function and produced greater ease of debugging. The modules could be incrementally tested and tasking made localization of failures both quicker and easier to resolve. Device dependencies are confined to the modules serving the device. Thus, wholesale changes to, say, the display equipment in use would require no changes to MSS and MTWX. Similarly, removal or suppression of the Teletype operation would entail

simple removal or "non-attaching" of the MTWX code.

The task modularity also allows more graceful failure of the CHAT subsystem during execution. Like most medium-to-large sized programming systems, the CHAT Monitor is liable to contain undiscovered latent errors; moreover, as the CHAT support grows, new errors may be introduced. Thus, it is useful in a system planned for production operation to take advantage of the OS/360 MVT facilities for partitioning and signalling task failures to allow orderly termination and possible restart.

MSS is the topmost task in the CHAT region task control hierarchy, being attached by the control program initiator and, in turn, attaching the other control tasks. This activity is performed during the initialization phase of the CHAT region and involves execution of code having only transient residence in the region. This initialization phase activity is sufficiently complex that the details are left to a separate section at the end of this Chapter.

The relative priority of the two remaining control tasks is a function of the order in which they are attached by MSS--obviously a simple matter to change. Currently, this order involves attaching MIOS first, then MTWX, which gives display I/O scheduling an intrinsically higher priority than Teletype I/O scheduling.

MSS continually attaches, task-manages, and detaches application programs. Thus, MSS is the "owner" of all tasks in the CHAT region. This design involves coordination between the other control tasks and MSS, both for task initiation and, at times, for task termination. MIOS, for example, first detects the change from inactive to active status of a display by detecting an interrupt from it. Similarly, MIOS may also detect improper behavior in an application task's usage of the

display interface that calls for task dismissal from the system. Each need is communicated to MSS by MIOS through the post/wait protocol. Conversely, MSS occasionally needs to signal MIOS or MTWX of an abnormal termination of an application task so that I/O termination procedures (e.g., the display of the "Proctor message") can be taken. Chapter 4 discusses the advantage MSS takes of the OS/360 task management services to do scheduling of CHAT applications.

Core Layout and Intraregional Protection: The core layout of the CHAT region is interrelated with intraregional protection. Hence, some background is needed on the protection mechanisms of the CHAT environment.

The System/360 protection scheme is quite simple. Each storage block may have one of sixteen storage keys (0-15). Read/write access is governed by a matching of a key in the program status word (PSW) with the storage key on execution of instructions involving core storage. A similar mechanism governs channel accesses to core. This is enough storage keys in an environment involving solely batch-type multiprogramming in which task and job are, in practice, naturally synonymous. Where jobs support interactive computing from a multitude of terminals this number is no longer satisfactory, and the limit of one nonzero key per job imposed by OS/360 is utterly inconsistent with the tasking concept. This shortcoming does not appear in the Dennis and Van Horn definition [D2].

Interactive computing from a multitude of terminals usually involves a number of autonomous program-execution activations, or threads, equal to the number of connected terminals. Multithreading requires control program support for both thread-switching and protection of threads from

violation by other threads. Multithreading can be realized by a number of mechanisms. Strictly interpretive systems (e.g., CPS, APL), whereupon all user programs execute on a closed high-level machine, can be built upon any one of the variants (MFT, MVT) of OS/360. Here thread-switching and protection are features of the higher-level machine. Other interactive support systems such as IBM's Time-Sharing Option (TSO) [S1], RTOS [W2], and CHAT are extensions to OS/360 MVT that do multithreading by exploiting the control program multitasking services, and must somehow compensate for OS/360's omission of complete individual task protection. Let us quickly review these techniques.

TSO uses an approach with performance complications and couples storage protection with the mechanism of swapping provided for program storage management. Each terminal thread is treated as a separate job and multiple jobs share the use of a storage region (and a common key). When job switching occurs the old job is rolled out and the newly active job is rolled into the region from secondary storage. To enhance performance, a number of regions dedicated to TSO can be specified. A degenerate form of the TSO method, avoiding swapping, is to allocate one region per terminal; but this approach suffers from the OS/360 limit on the maximum allowable number of regions. Note that because a fresh copy of a program sharing a key with another must be fetched from secondary storage prior to its activation, the TSO approach is not able to take full advantage of unlimited storage availability.

RTOS takes a different tack. There, the notion of "independent task" is defined. (Recall RTOS also defined an unrelated "system task"). Creation of an independent task results from an RTOS-defined RTATTACH rather than through use of the standard OS/360 ATTACH (which RTOS also

allows). Thus, RTOS extends the OS/360 MVT task creation facility to give independent protection to "independent tasks" within a single region. What is not clear from the available documentation [W2] is how the task-switching facilities of OS/360 MVT are also modified to achieve the promised intraregional protection.

A possible implementation would be to modify the task-switching mechanism to allow the sharing of an "active" (nonzero) key. The deactivated task (and possible subtasks) could have all its storage changed to an "inactive" key, while the storage keys and task key of the activated task (and subtasks) could be assigned the value of the shared active key. This approach also has performance difficulties since storage key changing, particularly where storage is fragmented, is not notably efficient in the System/360 (a fact that could lead an implementation to share a "pool" of active keys to improve performance).

Furthermore, I/O realities represent a complication for such a design. A deactivated task can have pending, or on-going, I/O operations which are also governed by the storage key mechanism of System/360. In RTOS, all I/O is governed by an access method (RTAM) occupying its own separate region (and storage key); and storage buffering for I/O is centralized in this region. I/O is a problem in TSO as well. There, the telecommunications I/O is also centralized and performed in "fixed" storage rather than in the "swappable" storage, because of its intrinsic slowness relative to swapping rate.

The RTOS approach, whatever its implementation difficulties for general-purpose support, embraces a generalization that probably should have been included in OS/360 MVT--with the protection attribute perhaps included as an additional operand in the standard ATTACH rather than

implicitly through a separate operator.

The CHAT implementation approach is to lessen the protection exposure without fully eliminating it. Swapping was clearly beyond the manpower and time constraints of the CHAT design effort, particularly since no primitives exist in OS/360 to facilitate designing such a facility. Modifications to control program facilities to achieve "independent tasking," à la RTOS, violates the basic constraint that CHAT would avoid release-dependent updating to crucial control program facilities. Finally, incorporation of the CHAT support in a non-dedicated multiprogramming installation limited CHAT's residence to a single region because of the OS/360 limit on the number of regions.

The CHAT support, of course, requires that the Monitor be protected from application program interference and that deviations or "bugs" in one application program not be allowed to destroy the operation of another application program. The approach CHAT takes to give protection is to make use of dual keys and storage "checkerboarding."

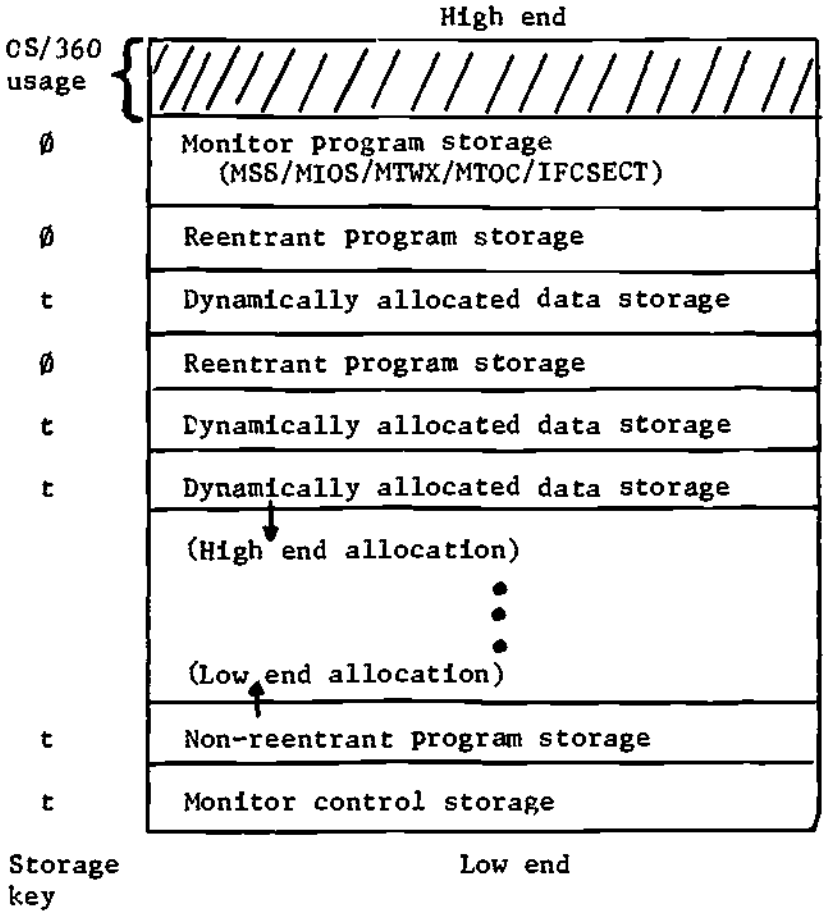
The Monitor program modules and some application programs intended for multiterminal use are marked as "reentrant" (to the linkage editor) and are thereby loaded into key-zero storage. Their task keys, which determine their PSW-keys, are all set to the nonzero key allocated by the control program to the CHAT region. Thus, no reentrant program storage in the CHAT region can be written into by an active task. Dynamically allocated data areas are resident in storage having the same (nonzero) storage key as the common PSW-key.

Another category of nonreentrant application programs exists. These programs generally include those application programs (e.g., PAMELA) mentioned in Chapter 1 that were converted to CHAT and which were ori-

ginally designed for single-terminal operation. They may be used for multiterminal operation, but because they are not marked reentrant, an individual copy of the program is loaded for each terminal invocation. Multiterminal usage of the reentrant application programs involves sharing of a single copy of the program since such programs must, in fact, be truly reentrant. The nonreentrant programs are loaded, generally with contiguous static data storage, into storage having the nonzero task key. Thus, program and data storage are modifiable.

The Monitor itself does violate certain reentrancy criteria. It cannot be shared to serve more than one CHAT multiplexer-display cluster and one of the Monitor tasks does indeed perform modification of program-storage. The sharing constraint is apparently not a drawback, since we do not anticipate multiple clusters. The program-storage modification actually involves a mechanism for lessening the protection exposure and thereby increasing the debuggability of Monitor failures. The pertinent Monitor task is MIOS, which does channel program generation and I/O within the MIOS program-storage area to isolate these activities more fully from attack by tasks. This effectively reduces the confounding of intrinsically complex hardware failures with CHAT region program anomalies. The task key of MIOS must be changed to zero during the course of such modification.

Figure 2.4 illustrates the core layout of the CHAT region with storage keys indicated. The OS/360 control program allocates storage starting from the high-address portion of the region for dynamically allocated storage and reentrant-marked programs and starting from the low-address portion for nonreentrant programs. As stated previously, the CHAT tasks operate with the single CHAT region nonzero key, with the



t: nonzero key assigned to the CHAT region (agrees with the task key)

Figure 2.4. CHAT Region Core Layout (in LCS)

exception of periodic excursions by MIOS into key-zero. Part of the initialization procedure to be discussed in the last section of this chapter, is to load the Monitor storage into the low-end of the region by marking it as nonreentrant "program" storage.

Notice that the storage areas that can be written into, i.e. those with the nonzero storage key, alternate with the key-zero storage areas, which are write-protected. This effect, for lack of a better metaphor, is referred to as "checkerboarding." The layout of CHAT storage varies with time and usage. The invocation of application programs is subject to terminal operator selection and, hence, Figure 2.4 is only one possible layout. Only the program and data storage for the Monitor are fixed. It is, in fact, possible that a nonreentrant program will be located above reentrant program areas, depending on core availability at the time of loading.

CHAT, then, does not offer full protection but its techniques do lessen the exposure. Programs that attempt to write improperly into non-owned storage within the region must hit on nonzero storage in order to do harm. Since such storage "misses" are more likely to be too high (particularly for those application programs coded in PL/I, where subscript errors, at least, are seldom negative-valued), the placement of the Monitor data storage at the low end of the region yields an additional protection advantage. A heartening measure of the degree of the threat is that such a protection violation has never been observed in experience with the CHAT System during its more than two years of operation. This is all the more remarkable considering that this period has seen a large number of newly developed programs introduced into the CHAT subsystem.

Despite this good report on CHAT operation using only the facilities available, one still wonders why the design of OS/360 ignores the requirements of the interactive computing environment for fuller protection.

Communication and Linkage: Modularity brings increased communication complexity. Within the CHAT region, communication is essential to intra-Monitor control as well as between Monitor and application tasks. The Monitor data storage area contains two primary divisions: the Monitor working storage and the storage occupied by the station control blocks (SCBs).

The Monitor working storage is addressable from all three control tasks and, with the previously cited exception of MIOS-controlled channel program areas, contains all constants and variables used during execution of the Monitor tasks. For the purpose of coordinating "regional events," e.g., region shutdown, control task failure, and Teletype-initiated display-complex reinitialization, certain event control blocks (ECBs) are defined within the working storage. Communication among the Monitor control tasks for such regional control uses these ECBs in conjunction with the OS/360-supported wait/post protocol.

The SCBs are used to hold all constants and variables needed for operation of the individual stations (displays and Teletype) in the CHAT configuration and for associating the appropriate application program subtask to the station from which it was initially invoked. The individual fields in an SCB include those used as event control blocks associated with various requests and outcomes connected with individual station and subtask operation, a field for holding the address of the associated application program task control block (TCB) for linking subtask to station,

and various other fields for input and output data, PL/I "dope vector" type information, and current I/O and subtask status conditions.

Association of station to application subtask is an involved process established initially at the time the first interrupt arrives from a previously idle display or when the "$XEQ" command is received from the Teletype. In either case, MIOS or MTWX posts an MSS-owned ECB in the corresponding SCB, which causes MSS to attach a module named MTOC (for Monitor table of contents routine) written by Blair [B3]. MTOC is reentrant and always resident, being packaged with the modules comprising the Monitor program storage shown in Figure 2.4. MTOC represents a "front end" to every application program that runs in the CHAT region. It is MTOC that causes the program table of contents to be displayed on the display CRT and decodes the operator's application-invoking request (in the case of the Teletype, the decoding proceeds at $XEQ-time, without an intervening "display"). MTOC sets up the name of the requested application program and issues an OS/360 XCTL (Transfer Control) macro operation. Thus, the invoked application program inherits the MTOC TCB.

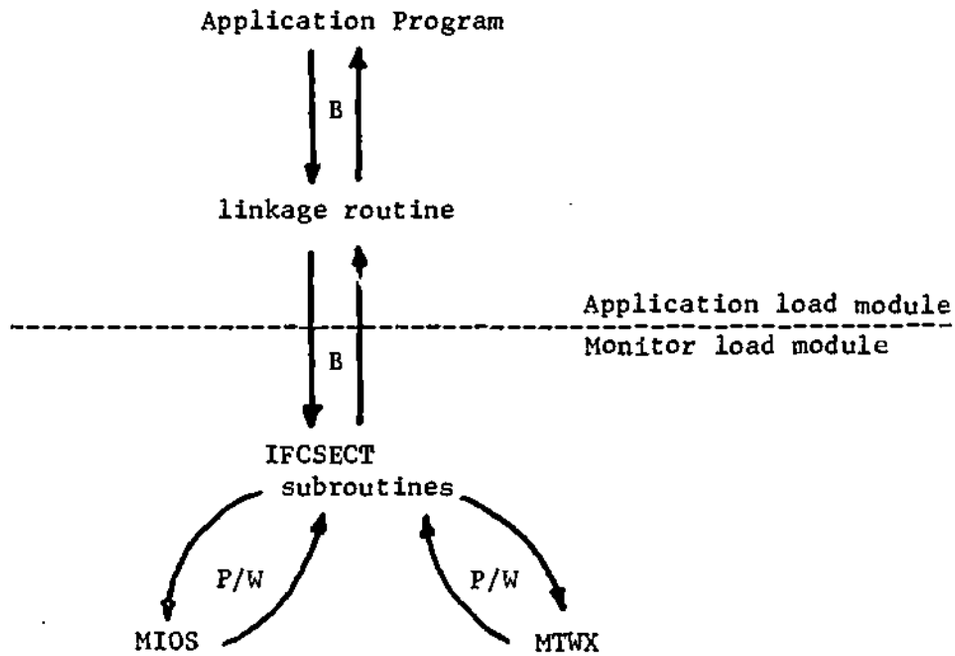
The purpose of the front-end MTOC is primarily to protect the Monitor. Under OS/360, if an attached program cannot be located or if storage is not available for it to be loaded, the attaching task is aborted. Thus, such circumstances have dire effect on MTOC but, under OS/360, are simply "reported" (via posting) to MSS. MSS can then initiate a proctor-message to inform the station operator.

Two other operations are performed during this MSS-attaching of MTOC: MSS, on completion of the OS/360 ATTACH macro, stores the OS/360-returned TCB address in a field of the SCB, while MTOC, via a Blair-

written SVC [B3], stores the SCB address in the so-called "TCBUSER" field of its own (and thus the application's) TCB. (The TCB is in key-zero storage--hence, the need for a "privileged" SVC to write into it.) The SCB address is passed to MTOC by MSS through a parameter-passing mechanism of the OS/360 attaching protocol. Storing of the TCB address in the SCB and of the SCB address in the application TCB effectively weds station to application subtask.

To hide this connection from application programs, rather than, say, including the SCB address as a parameter within the interface language (see Chapter 3), the linkage is reestablished upon every call of a Monitor-controlled function. This is done within an assembler-language coded interface routine linked within every application program load module. This "linkage routine" has three primary functions. The first is to select an entry point within a group of subroutines collectively packaged within another module (or OS/360 assembler language "CSECT") named IFCSECT which is not a part of the application program load modules but is packaged with the Monitor. (See Figure 2.5.)

A second function of the linkage routine is to select the appropriate SCB. This is done by a search through an MVT-defined control block chain shown in Figure 2.6. Finding the address of the SCB means that the address of the appropriate IFCSECT subroutine can also be located. A pointer exists in the SCB to a list of the subroutine entry points; entry to the linkage routine determines the index into this list. Before branching to the appropriate subroutine entry, the linkage routine sets up a register with the SCB address to give the subroutine addressability to SCB storage. When the IFCSECT subroutine posts an ECB for a Monitor control task, the address of the SCB forms the low-



B: Synchronous Branching
P/W: Post/Wait

Figure 2.5 Linkage Between Application and Monitor

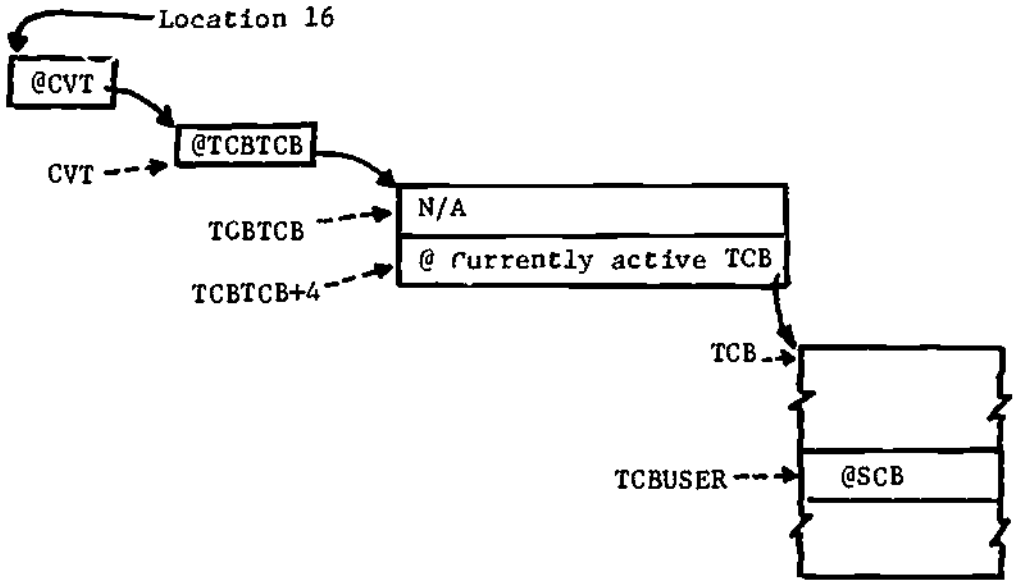


Figure 2.6 OS/360 MVT Chain to TCBUSER Field

order three bytes of the ECB fullword completion code. This passing of the SCB address via the ECB is followed throughout the Monitor during posting of station or application subtask related events.

The third function of the linkage routine is to set up addressability to a parameter list containing the PL/I-like "STOP" and "ABNORM-condition signalling" routines. In the case of application programs written in PL/I, these are the addresses of the actual PL/I-provided library routines. (See Chapter 3 for definition of the "ABNORM" condition.) Completion codes set by the Monitor control tasks in posting IFCSECT subroutine ECBs (also in the SCB) as well as IFCSECT subroutine detected errors in application program passed parameter lists cause these routines to be invoked.

In current practice any one of four different linkage routines can be linked in with the application program load module, depending on the intent of the application and the language in which it is coded. These are named IFNTRYS, IFNTWXS, OLNTRYS, and IFNTRYAS, the latter one of which was coded by Blair [B3] and the others by the author. The first three linkage routines are for use by application programs coded in PL/I. Each linkage routine has the same general structure. The PL/I program's execution of a call-statement or function reference specifying a Monitor-controlled facility (see Chapter 3 for a description of this interface) results in a linkage to one of the entry points in a multiple-entry branch table in the linkage routine. Each branch statement branches to a join-location while simultaneously initializing register 15 (the entry point register, itself--and the only free register at this point in the linkage) to a value indicating the entry point. At the join point, code common to all entries is executed, saving registers,

establishing code addressability, searching through the MVT control block chain (Figure 2.6) to load a register with the correct SCB address, and developing an index value from the value put in register 15 by entry into the linkage routine.

This new index is used in conjunction with an address obtained from a field in the newly located SCB. This address is a pointer to a list of IFCSECT subroutine addresses. The index is used to choose the appropriate subroutine for the linkage routine to branch to as its final step. Of course, certain other functions are performed by the linkage routine prior to branching to the subroutine, like setting up addressability for the subroutine to a parameter list containing such addresses as those of the PL/I library routines for STOP and SIGNAL. A simple validity check is also performed to insure that a Teletype application program has not been invoked from one of the displays or vice versa. (The Teletype SCB is "marked.")

The linkage routines vary in the access they allow to IFCSECT subroutines. This is achieved by tailoring the multiple-entry branch table to the subroutine access required--a need, in turn, determined by the CHAT interface calls and function references included in the application program. IFNTRYS, for example, is the linkage routine provided when the application program includes the display and conduit interface. IFNTWXS is provided for Teletype application programs. OLNTRYS is a special-purpose linkage routine that permits access to both display and Teletype subroutines. It is used solely by a Teletype-invoked on-line test application program (OLTEST). OLTEST is the only application program that is permitted access to both the Teletype and the displays. Further discussion of this program along with the additional functions provided to

allow its unique interface privileges is deferred to a later chapter.

IFNTRYAS is the linkage routine provided for application programs coded in assembler language. It is invoked by assembler macro-instructions having a syntax nearly identical to that provided by the CHAT interface for the PL/I programs. These macro-instructions are described by Blair [B3] who defined them. One important user of the assembler interface is MTOC which uses it, for example, for the display of the program table of contents on a display station CRT. (MTOC does not do Teletype I/O.)

IFNTRYAS is similar to IFNTRYS in being display-access oriented. It, in conjunction with Blair's macros, has certain additional responsibilities for creating PL/I-like environment characteristics, such as imitating the "dope vector" conventions and duplicating the STOP and SIGNAL capabilities. These matters are discussed further in [I10]. IFNTRYAS is like the other linkage routines in its branching to IFCSECT subroutines; thus, the semantics of the assembler language macro interface is identical to that of the equivalent PL/I interface. This linkage technique could very well be extended to allow the interfacing of application programs written in still other languages besides PL/I and assembler.

The IFCSECT subroutines are reentrant and are packaged in the Monitor program load module. This means that their program storage overhead appears once in the CHAT region, with all currently resident application programs sharing their use. This is in contrast to the linkage routines, which are duplicated for each copy of an application program in residence.

The IFCSECT subroutines operate under control of the same TCB used by the application program whose linkage routine branches to them.

Depending upon the function requested by the application program, these subroutines may represent the final or a still-intermediate stage of linkage in carrying out the semantics of the application program request. Some requests can be honored by a simple translation or by an accessing of a field in the SCB; others involve more complicated communication with one or more of the Monitor control tasks to initiate an I/O operation and to await its signalled completion.

Communication between IFCSECT subroutines and the Monitor involves both SCB storage and the post/wait protocol. Parameters passed from the application program via dope vectors are placed in the SCB by the subroutines to control the Monitor's fulfillment of requested operations. Information stored in the SCB by the Monitor is used by the subroutines to update the dope vectors and application data storage prior to returning to the application program. Each SCB also contains a number of fields used as event control blocks (ECBs). Certain ECBs are owned ("waited upon") by the subroutines, while others are separately owned by the various Monitor control tasks. (Joint ownership is impossible.) These ECBs, like Dijkstra's semaphores, are used in synchronizing subroutine and Monitor by post and wait coordination. The subroutine requests an operation by initializing a request-type field and then posting a Monitor-owned ECB; the Monitor signals a completion by posting the subroutine-owned ECB.

Variations to this simple protocol are also possible. For example, the Monitor task, MIOS, is designed to act on display interrupts independently of a pending application request for input. In the case of a lightpenning action, the Monitor may have completed the entire sequence of obtaining lightpen coordinates and lightpenned character, including

storing the information in the SCB, prior to the application program's official request. In such a case the subroutine will detect by a "status" field in the SCB that communication via post/wait is not needed. Instead, the information can be immediately retrieved, the "status" modified to reflect this pickup, and control returned to the application program.

Still another variation involves synchronization constrained by time, where an application program has issued a read operation with a time limit (see "PAUSE" in Chapter 3). In this case, use is made of an asynchronous IFCSECT "timer completion exit routine" (a coroutine of the IFCSECT subroutine). Here, if the SCB status does not show the input to have already occurred, the subroutine will initialize the request-type field in the SCB, post an MIOS ECB, issue a timer request (via OS/360 "STIMER") specifying the exit routine, and wait on two ECBs. One can be posted by MIOS, the other by the exit routine. When one of the ECBs is posted, the IFCSECT subroutine regains control and, after determining which event occurred, cancels the other request. Cancellation of the timer request involves (for PAUSE) obtaining the amount of time expended, for the purpose of relaying this information to the application program.

For operations involving I/O, the IFCSECT subroutines are concerned primarily with signalling the request, depending on the target station, to either MIOS or MTWX and then synchronizing the reactivation of the application with the posted completion. Input requests involve a slight elaboration to allow additional communication with MSS. Here, the subroutines set an "idle-flag" in the SCB prior to awaiting completion of the input request. This allows MSS to remove the application subtask from its queue of dispatchable subtasks to reduce execution overhead during its time-slicing and subtask scheduling epochs or to suppress

such epochs entirely if all application subtasks are idle. This allows the CHAT region to quiesce entirely during protracted periods of idleness.

On completion of such input, the reactivated IFCSECT subroutine posts an MSS-owned ECB in the SCB to alert MSS of its newly-regained dispatchability. MSS can then include the application subtask once again in its scheduling process before the subroutine returns to the application program.

Subroutine requests to the Monitor control tasks result in considerably more complicated activity within these tasks, depending upon the current state of the transmission line and the existence of other outstanding requests. The queueing and scheduling techniques employed are described in subsequent chapters dealing with these components of the Monitor. Monitor I/O completion posting can involve three types of indicators for the subroutine: proceed normally, invoke an application program "on-unit," or stop the application program. Hence, the reactivated IFCSECT subroutine can take any one of three return exits. The subroutine can, itself, initiate the "on-unit" return when an application program's request-parameters are faulty. Specific conditions causing these various exits are also described in subsequent chapters.

INITIALIZATION OF THE REGION

Initialization of the CHAT region is a complicated multistep process. Its primary objectives are to produce the initial regional layout of the Monitor indicated in Figure 2.4--the Monitor control storage at the low end, and the Monitor program load module at the high end--and to prepare the resident control tasks to perform their monitoring activities during subsequent CHAT System usage. To prepare the control tasks, it is necessary to give them addressability to the control storage, to initialize the control storage so that all required run-time parameters and linkages are set up for use, and to perform any once-needed start-up initialization of the equipment. A design goal, that none of this initialization code would be permanently resident in the CHAT region, complicated the sequencing and linkages of the initialization steps. The design of this step sequencing and linkage was the joint effort of the author and Blair, although Blair's inspiration dominated.

Figure 2.7 gives a gross view of the initialization steps. The type of linkage involved from step to step is shown in parentheses along the arrows; the other names are those of the actual code steps invoked. (ROOTCAI AND INITCAI were named before the acronym CHAT was contrived, when the CAI Project was foremost in mind; better names would be ROOTCHAT and INITCHAT.) The functions of the various steps are briefly as follows:

ROOTCAI - This small root of code is assembled in the low end of the Monitor control storage load module. It is attached as the CHAT job step task (cf. Chapter 4)--an identity in-

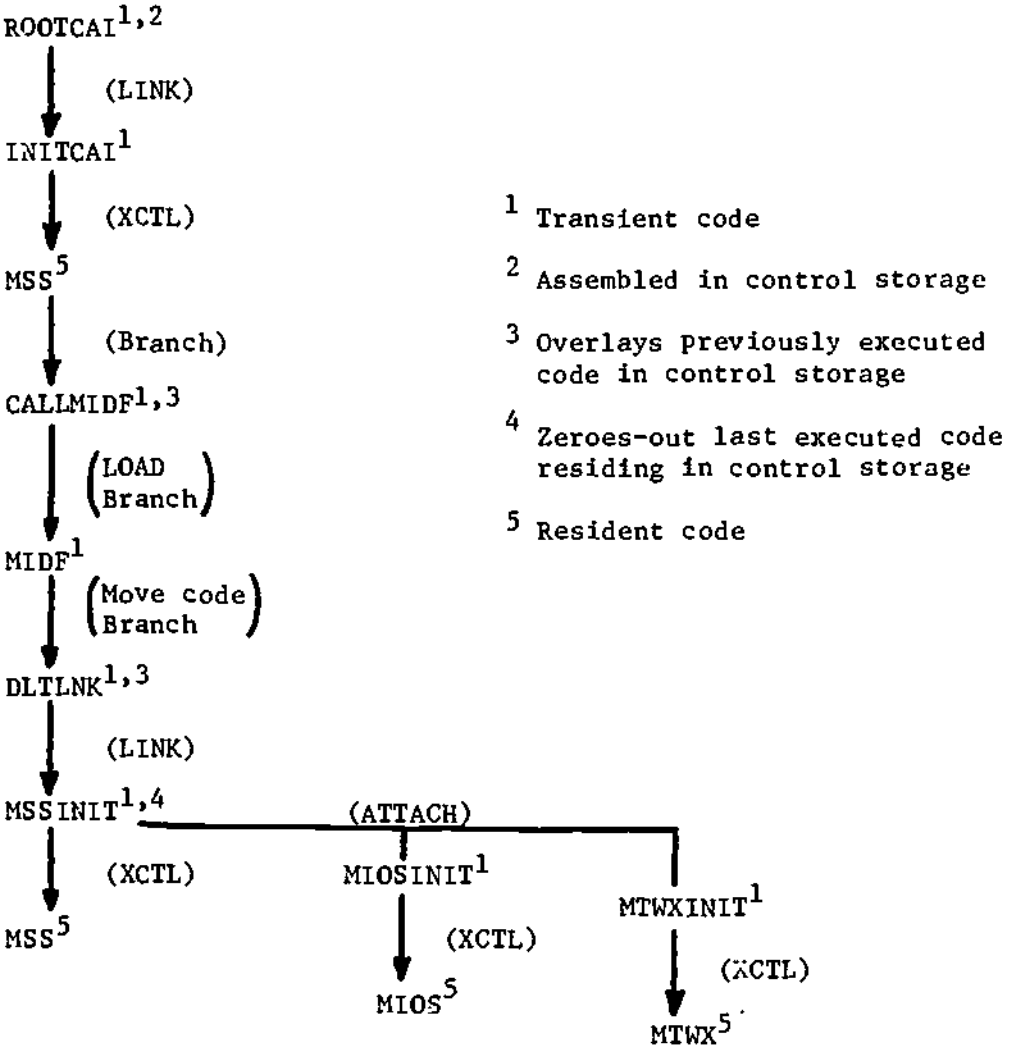


Figure 2.7. Initialization Steps

herited by MSS via the initialization linkage. ROOTCAI serves two important purposes: (1) Because ROOTCAI is part of the control storage load module and is marked as a non-reentrant program, OS/360 loads the Monitor control storage at the low end of the region; (2) ROOTCAI passes the address of the control storage through the subsequent linkage protocols to the Monitor control tasks. It uses the OS/360 LINK macro to pass control to INITCAI.

INITCAI - This step passes addressability to Blair's "SVC 239" [B3] and loads the OS/360 abnormal dump resident module (also described by Blair [B3]) and moves the CALLMIDF code into the Monitor control storage, overlaying ROOTCAI. It then uses the OS/360 XCTL macro to transfer control to MSS, which in the process causes OS/360 to load the Monitor program load module into the high end of the region.

MSS - After some standard establishment of addressability (needed for run-time operation), MSS branches to the CALLMIDF code in the control storage, passing the address of an entry point table identifying the locations of other components of the Monitor program load module--a list of the IFCSECT subroutines, MIOS, MTWX, MTOC, and the MIOS Start I/O appendage.

CALLMIDF - This step begins some "middle functions" necessary to initialization. It uses the OS/360 LOAD macro to load MIDF and then branches to MIDF.

MIDF - This step continues the middle functions: it uses the OS/360 IDENTIFY macro to make MIOS, MTWX, and MTOC known as entry points to OS/360, stores the address of the entry point table in control storage, and also moves the DLTLNK code there, overlaying CALLMIDF. It then branches to DLTLNK.

DLTLNK - This step invokes an OS/360-supplied SVC to delete MIDF (deleting the linkage as the acronym implies) and uses the OS/360 LINK macro to branch to MSSINIT.

MSSINIT - This step initializes the station control blocks (SCBs) in control storage with the address of the list of the IFCSECT subroutines so that the application program entry linkage invoked by the application interface (described in Chapter 3) can find the subroutines. It also initializes the portion of control storage used by MSS: it clears the entry point list address and DLTLNK code moved there by MIDF and stores task-priority increment/decrement values used by MSS for time-slicing and application subtask scheduling. This initialization involves searching through the CVT-chain (shown in Figure 2.6) to find the active TCB which contains the current run-time priority ceiling for CHAT. The final function of MSSINIT is to attach MIOSINIT and MTWXINIT (in that order) which operate as separate Monitor control tasks. MSSINIT then XCTLs to MSS at an entry point (MSSWAIT) which waits for a run-time event to occur.

MIOSINIT - This is the most complex and involves the most code of all initialization steps. Chapters 5, 7 and 8 give additional details; here we mention only that it initializes the inter-regional conduit and prepares the display equipment for run-time use. When done, it XCTLs to MIOS, which inherits its task identity as one of the three Monitor control tasks.

MTWXINIT - This step prepares MTWX for its run-time operation of the Teletype. Besides initializing some control storage fields, it translates from EBCDIC to Teletype code the text of messages MTWX sends to the Teletype in support of its command functions. It uses the OS/360 OPEN macro to initialize the Teletype control block needed for input/output operations and provides MTWX addressability to the OS/360 Unit Control Block (UCB) needed for a Halt I/O function it performs. When done, MTWXINIT XCTLs to MTWX, which inherits its task identity.

Because of the linkage conventions employed, when MSS, MIOS, and MTWX are finally initialized and ready for CHAT System operation all traces of the described initialization steps are removed from the region. This includes even the linkage to them, with the unavoidable exception of two instructions in MSS for branching to CALLMIDF. As an illustration, however, that CHAT, like most systems, violates its purity of intention, MTWX contains a small amount of code to put the addresses of its two "channel end appendages" in an OS/360 control block prior to beginning its operation. This deviation was a conscious tradeoff

favoring time and resources (and sloth) over purity and further complexity. (MTWX was coded last.)

CHAPTER 3: THE APPLICATION PROGRAM INTERFACE

This chapter describes the subroutines and functions that may be invoked by the application program in order to access Monitor-controlled facilities such as display and Teletype input and output and the conduit to CPS. For an application program coded in PL/I, these procedures are invoked by means of call statements and function references. The syntax and semantics of these procedure references are discussed in sufficient detail to allow this chapter to serve as external specifications for the PL/I application programmer interested in writing a program for the CHAT system.

A simple metalanguage is employed in presenting the procedure references: underscored referents represent arguments; alternative coding formats for generic procedures are shown by stacking the optional versions within braces. Names preceded by the "$" symbol are not part of the metalanguage but are actual CHAT variable names defined in declarations which the programmer will normally have generated at compile-time by use of the %INCLUDE preprocessor statement. The values of the $-named variables are purposely not shown here. If the programmer desires this information, it can be obtained by examination of a source listing containing these compile-time included declarations. Use of this compile-time facility is discussed in more detail in a later section of this chapter.

To illustrate the metalanguage, suppose a fictitious CHAT procedure, PROCNAME, were provided with the metalinguistic representation:

CALL PROCNAME({variable1,variable2 / $SYMBOL}) .

In this case, the description (following the metalinguistic representation) of PROCNAME would give attribute and range (or value) requirements for the variable1 and variable2 arguments. If the two-argument format were used, the programmer would declare and set two variables of his own choosing, say ARG1 and ARG2, and would code:

```
CALL PROCNAME(ARG1,ARG2).
```

The second format would be coded as shown:

```
CALL PROCNAME($SYMBOL)
```

where $SYMBOL would be declared in the compile-time included text provided by CHAT.

The procedures discussed here can also be invoked by application programs coded in assembler language, using a set of assembler macros designed by Blair [B3]. These macros are similar syntactically to the PL/I procedure references described here and imitate the PL/I internal linkage and parameter-passing conventions sufficiently that the invoked procedures are insensitive to the source language of the invoking program. The invoked procedures are themselves coded in assembler language.

DISPLAY USAGE

Procedure references are provided to display text on the display screen, to read data that the display operator has keyed or light-penned and to control the slide projector connected to the display station output channel.

Controlling the Slide Projector: There are two different types of slide control offered to the programmer: turning the slide projector power on or off and selecting a specific slide for display. These are accomplished by means of the generic procedure reference SLIDE (a difference in number or attribute of allowed arguments requires generic definition in PL/I):

```
CALL SLIDE( { $ON          } ).
          { $OFF         }
          { slidenumber  }
```

$ON and $OFF (both BIT(1) variables) perform the functions to which they mnemonically refer: slidenumber is any FIXED BINARY(15,0) variable in the range 0-80 corresponding to the carousel slide position to be selected. The programmer need not explicitly code the power-on function: the invoked subroutine will turn power on automatically if a carousel motion is requested when the projector has power off.

The experienced display station user is familiar with the fact that, while the slide slots are numbered in base-10 representation on the carousel, keyboard control of the slide equipment requires base-9 representation. This is true also for computer control of the slide equipment (where the base-9 codes must also be sent in the ASCII format). However, the application programmer need not worry about this vexing base-9 control;

CHAT transforms the application program base-10 representation to base-9 (in ASCII) automatically.

On completion of a slide operation, the cursor location and keyboard state (locked/unlocked) are the same as prior to the SLIDE issuance.

Writing on the Display Screen: To display alphanumeric text on the display screen, the programmer uses the generic procedure reference DISPLAY:

$$\text{CALL DISPLAY}\left(\left\{\begin{array}{l}\underline{\text{message}}\\ \underline{\text{message}},\underline{\text{row}}\\ \underline{\text{message}},\underline{\text{row}},\underline{\text{col}}\end{array}\right\}\right).$$

The row and col arguments are FIXED BIN(15,0) variables specifying where the cursor is to be positioned prior to the display of the message character string. The absence of the row and col arguments in the top format means the message character string is to be displayed starting at the current position of the cursor, that is, wherever it was left at the completion of the last operation. As explained below, the message argument can contain embedded format controls to control cursor placement, thereby obviating the need for the row and col arguments. The middle format above, with col absent, is taken to imply the first position (column) of the specified row. The value represented by the row argument must be a legal row number; i.e., in the range 1-20; similarly, the col argument must represent a value in the range 1-40. The message argument must represent a character string of length 0-810 characters, a limitation to be discussed shortly.

Table 3.1 contains a list of format controls: $-named variables and a function reference, LCAR, that represent format characters, which, when sent to the display, control cursor placement, keyboard enabling,

Cursor Control:	
$C	Clear display screen; place cursor at first position of first row (CLEAR).
$R	Return cursor to first position of current row (RETURN).
$U	Move cursor up one row (↑).
$D	Move cursor down one row (↓).
$B	Backspace cursor one position (←).
$F	Forespace cursor one position (→).
$S	Move cursor to first position of first row (RESET).
$T	Move cursor to 21st position of current row (TAB).
$L	Display New Line symbol (◢) at current cursor position and move cursor to first position of next row (LINE).
$DR	Same as $D\|\|$R (↓,RETURN).
$DDR	Same as $D\|\|$D||$R (↓,↓,RETURN).
LCAR(row,col)	A function reference for "random" cursor placement; cursor to be moved to position with coordinates specified by the FIXED BIN(15,0) row and col arguments.
Keyboard Control:	
$E	Enable (unlock) keyboard (MASTER CLEAR).
Color Selection:	
$GRE	Select green (SPECIAL CODE + Q).
$RED	Select red (SPECIAL CODE + R).
$BLU	Select blue (SPECIAL CODE + S).
$YEL	Select yellow (SPECIAL CODE + T).

Table 3.1: Format Controls

and color selection.

In the table, the keycap names corresponding to the functions are shown in parentheses. Experimentation with an actual keyboard will best familiarize the reader with the functions described. An equivalent keyboard function for LCAR does not exist; while a four-character control character sequence sent from the computer provides the function, it can only be accomplished at the keyboard by compound use of the other keys.

The format characters themselves are eight-bit unprintable characters in EBCDIC. The $-names in the table, with the exception of $DR and $DDR, represent one-character controls; $DR and $DDR represent two- and three-character controls, respectively. The function reference LCAR causes a four-character string representing control of "random" cursor placement to be returned to the point of invocation within a message string to be sent to a display. Since the actual control characters required by the display equipment are foreign to the PL/I programmer (they are not only unprintable in EBCDIC, but require a Monitor translation to ASCII code when transmitted), the format controls were invented to ease the programming task.

Two constraints are imposed on the use of the format controls: if used, $C must appear as the initial character of the message argument; while $E, if used, must be the final character. If either rule is violated, the function is not performed; instead a blank character is sent in place of the misplaced character. Some illustrative examples and further discussion should clarify the use of DISPLAY and the format controls.

```
EX1:  CALL DISPLAY($C||'EXAMPLE MSG'||LCAR(20,1)||
                   'ENTER CODE'||$F||$E);
```

The above statement causes the display screen to be cleared, displays EXAMPLE MSG on the top line, positions the cursor at location

(20,1), displays ENTER CODE there (the bottom line of the screen), moves the cursor one more space (leaving it at location (20,12)), and enables the keyboard. Enabling the keyboard need not be done explicitly in this way: the Monitor will automatically enable the keyboard on issuance of a subsequent read-type call statement that follows a DISPLAY where $E is not used. When $E is omitted, the keyboard is left in the locked state at the completion of the display operation. The point of providing the explicit function is to permit the keyboard to be unlocked immediately upon completion of a display operation that is to be followed by a read-type operation. In this manner, delays in issuing the subsequent read are not noticed by the station operator.

```
EX2:  CALL DISPLAY($BLU||MSG1||$DR||$RED,R10);
```

The above example demonstrates the use of color formatting. The character string represented by MSG1, appropriately declared elsewhere in the program, is displayed in blue starting at the first position of row ten, assuming that R10 is given this value of ten elsewhere in the program. The cursor is left at the first position of the row following the row displaying the last character of the MSG1 string. Because $RED is used, upon completion of the operation the cursor is displayed in red and, if a read-type operation is next to occur, any keyed-in characters will appear on the screen in red, thereby distinguishing computer responses from operator input. Notice that the above example, by omitting the $E control, leaves the keyboard locked.

Earlier, the length bounds for the message argument were given as 0 and 810. The reader may wonder what a DISPLAY with a null message argument does. The answer is that it simply locks the keyboard. At the other extreme, 810 characters is sufficient to permit the display of a

full screen, 800 characters, and to allow up to ten control characters as well. An upper limit is necessary because of the fact that the Monitor uses an intermediate buffer into which the message argument string is transferred prior to being transmitted over the communication link. In the unlikely case that the programmer needs to define a message argument of length greater than 810, the recourse is to use more than a single DISPLAY statement.

If at the completion of a DISPLAY, the programmer wishes to determine where the cursor was left, he may use the CURSOR call statement:

CALL CURSOR(row,col)

The invoked subroutine returns the cursor location in the FIXED BIN(15,0) row and col arguments. This call statement, if issued after a READ call (discussed below), will return the position of the cursor as it was left at the end of that operation.

Reading from the Display: To read data that the display operator has keyed in at the display station, the programmer uses the generic procedure READ:

CALL READ($\left\{\begin{matrix}\text{message}\\ \text{message,row,col}\end{matrix}\right\}$)

The FIXED BIN(15,0) row and col arguments specify the starting position on the screen for the read operation. In the top format, where these arguments are omitted, the starting point is assumed to be at the position where the cursor was left at the end of the last operation, whether this was a DISPLAY or another READ. The message argument represents the location where the received message string is to be returned by the invoked subroutine. It may have maximum length of the programmer's choice (including zero). Providing a maximum length shorter than the

keyed-in string causes a truncated message to be returned. The Monitor will read from the specified (or implied) starting point through the position preceding the location which the cursor holds at the time the display operator presses the interrupt (INT) key. This assumes the final cursor location (at interrupt time) is greater than the starting location. If this order is reversed, the Monitor will read from the starting location to the last position on the screen.

In either case, the issuance of the CURSOR call following the READ will result in the coordinates the cursor held at interrupt time being returned. The Monitor not only restores the cursor to the position it held at interrupt time, but restores the character (typically a blank) over the cursor, since reading from the display involves temporarily placing an "end" character (ETX) after the final data position.

Getting the Lightpenned Coordinates: To read the location of the search-indicator that illumines the background of a lightpenned character, the programmer issues one of two procedure references:

```
CALL READLP(row,col)
CALL LPLOCN(row,col)
```

In either case, the coordinates of the lightpenned position are returned in the FIXED BIN(15,0) row and col arguments. If the display operator has pressed the INT key without lightpenning something, row and col are set to (0,0). Otherwise, the value returned in row will be in the range 1-20; the value returned in col, in the range 1-40.

Before explaining why there are two variants for lightpen-reading, it is necessary to discuss the general notion of reading from the display, a discussion that applies to the READ operation as well. Reading from the display is not like reading a tape record: creation of a display

"record" is a dynamic thing and at the time of program readiness to read, the record may not yet be composed by the display operator. We need the interrupt signal, in general, to inform us that such a record now exists. Thus, the Monitor initiates the actual read operation only in response to an interrupt signal received from the display operator. Such an interrupt can, in fact, precede the program issuance of a read-type call, in which case, the read is initiated immediately upon issuance.

This synchronization of read operations (and also PAUSE, described below) with the interrupt signal means that on issuance of such an operation the keyboard must be enabled (unlocked), allowing the operator to key in data and to use the interrupt key. For operations, then, that are interrupt-responsive (read operations and PAUSE), the Monitor always enables the keyboard, if this is necessary. Where such an operation has been preceded by a DISPLAY using the $E function, the keyboard is already enabled and the Monitor need not again perform this function. If an interrupt has been received prior to an interrupt-responsive call, the enabling operation is not necessary: the read operation is ready to be performed.

This means that the Monitor is aware of occurring interrupts even if the application program has no call outstanding. The Monitor remembers an interrupt by setting an internal flag, which, for discussion purposes, we here refer to as the I-flag. In the case of a light-penning interrupt, the Monitor reads the lightpenned coordinates and the lightpenned character, as well. Prereading of a message cannot, of course, be done: the starting point is unknown until defined by the application program.

Issuance of a SLIDE, DISPLAY, READ, or READLP call-statement resets

the I-flag; LPLOCN does not. SLIDE and DISPLAY reset it because it is assumed that a subsequent read must receive a message which is conversationally responsive to whatever information these operations imparted to the display operator. READ and READLP reset the flag because they service the associated interrupt.

Herein lies the need for LPLOCN: the program may want the starting location of a READ to be defined by the operator in a dynamic fashion, for example, in the case of a text-editing operation. Lightpenning is a convenient way for the operator to perform this function. Hence, by first issuing LPLOCN, the program may then issue a READ with the row and col arguments set to the values returned by the preceding LPLOCN and service the same interrupt the LPLOCN responded to. This saves multiple interrupts at the display station.

After a successful lightpen reading the program may want to know the character that was lightpenned. This is achieved by use of the LPCHAR function reference:

```
EX:  X = LPCHAR;
```

where in the above example the lightpenned character is returned at the indicated invocation of LPCHAR, and in turn sets X to the desired character value.

Time and Keyboard Synchronization: Obvious programming difficulties can arise in reading from the display: the program cannot always anticipate the display operator's response. Will he lightpen something or key in some data before hitting interrupt? Or will he, in fact, do anything at all when the program has a read pending? If a READ operation is completed by a lightpenning response, the Monitor returns a null message;

if a READLP or LPLOCN completes because of a message-associated interrupt, the Monitor returns zero values for lightpen coordinates. These are indirect means the program has of detecting the mismatch between program and operator activity. However, to deal with these difficulties more directly and also to impose a limit on operator tardiness, the program may first issue the PAUSE procedure reference:

CALL PAUSE(time,return).

The FIXED BIN(31,0) time argument specifies the maximum nonnegative time interval, in seconds, that the program desires to wait for the display operator to hit interrupt. At completion of the operation, the BIT(8) return argument is set by the invoked subroutine to one of the following values, where as before the $-names are part of the compile-time included declarations:

$TIM — no interrupt has been received within the specified interval.

$LP — an interrupt has been received with lightpenning indicated.

$INT — an interrupt has been received without lightpenning indicated; a message is ready to be read.

Also on completion of the operation, the time argument is set to the time elapsed from the time the PAUSE was issued to the time the interrupt was received. In the case where $TIM is returned, the time argument obviously is unchanged, since the entire interval has elapsed. In the case where an interrupt has been received prior to the PAUSE issuance, time will be set to zero, since no time has elapsed.

A peculiar interpretation is given to a PAUSE that is issued with

the time argument representing a zero value: the maximum interval to wait for an interrupt is taken to be infinity. Here, only $INT or $LP can be returned; the time argument is left at zero on completion of the PAUSE.

The PAUSE procedure, like LPLOCN, does not cause the Monitor to reset the I-flag (if on). A read operation, following a PAUSE for which $LP or $INT was returned, will be executed in full immediately upon issuance and does not require a new interrupt to be received.

TELETYPE USAGE

Programming input and output for the Teletype is done by use of the two procedure references:

```
CALL WRTWX(outarea)
CALL RDTWX(inarea)
```

The outarea argument must be a character string of length 0-80. Each WRTWX automatically causes the carrier to be positioned to the left margin of a new line before the outarea character string is printed.

The inarea argument can have up to 80 characters returned, depending on the length attribute defined by the program for inarea and the amount of data keyed by the Teletype operator. Each time a RDTWX call is issued, the Teletype operator is alerted by the printing of the "?" symbol at the first position of a fresh line. Appropriate control (X-On) is also provided to allow automatic reading of paper tape. The Monitor strips out any control characters, such as carrier return or line feed, before presenting the keyed message to the program. Similarly, editing is performed to handle character corrections the operator has indicated during his keying operation through use of the underscore (or, on some Teletypes, the back-arrow) key.

CPS ACCESS

The procedure references provided for application program access to the CPS conduit are described here. Both the description and, more importantly, the use of the access facilities presume a knowledge on the part of the programmer of the manner in which he would interact with CPS if he were accessing that program from a CPS teletypewriter terminal. Other readers are referred to reference [11] for details on the CPS terminal operation.

Internally, the conduit employs the Teletype interface to CPS with the added capability for exchanging lower-case characters. Details of the internal design of the conduit are given in Chapter 7; here we describe the programming interface for using it.

Establishing Connection to CPS: To establish connection with the CPS facility, the program must issue one of the following versions of the generic procedure reference LOGCPS:

```
CALL { LOGCPS
     { LOGCPS(libname)
```

The no-argument format causes a "standard" log-in to be performed; sign-on identifiers are supplied by the Monitor. The second format above uses the one- to six-character libname argument to specify a load/save library identifier for this sign-on. When connection has not yet been established, the application program must issue one of the LOGCPS calls prior to using one of the calls described below.

Reading from CPS: To read output from CPS, the program uses the procedure reference RDCPS:

CALL RDCPS(inarea,time,return)

The inarea argument must be declared as a VARYING character string with length of the programmer's choosing. CPS output is truncated on the right if it is longer than the defined maximum for inarea. CPS messages are presented to the application program with all line and terminal control characters stripped out and in normal EBCDIC representation.

The FIXED BIN(15,0) time argument specifies the maximum time interval, in seconds, that is to be allowed for completion of this operation. The invoked procedure enforces a minimum specifiable delay of one second; an instantaneous completion is not possible because computer operations are not infinitely fast. (In fact, CHAT itself contends with CPS for use of the CPU--perhaps locking out CPS for brief periods.) Hence, a request of zero seconds is treated as one.

Upon completion of the operation, the BIT(8) return argument is set to a value represented by one of the following $-names:

$TIM — CPS has not responded within the interval specified by the time argument.

$MSG — the inarea argument contains the message character string returned by CPS.

$NULL — CPS has no output, but instead is ready to read from the application program.

The action of the application program following the completion of RDCPS is a function of the value returned in the return argument. Typically, if $TIM is returned, the program may try another RDCPS or an ATNCPS, discussed below. After $NULL, it is likely that a WRCPS will be

issued. In the case of $MSG, the application program's next action will likely be determined by the content of the returned character string, the analysis of which is the responsibility of the application program.

Writing to CPS: To send text to CPS, the application program uses the procedure reference WRCPS:

CALL WRCPS(text)

The text argument is a character string of varying length not longer than 256 characters, a limit imposed by CPS.

Because the CHAT interface provides no explicit logout procedure, the application program can achieve this by sending a logout request using WRCPS. The logout format is described in [I1] where different versions are shown. Notice that CPS responds with run-time statistics; these can be read by the application program in the usual way using RDCPS. If the application program uses the logout/resume variant, the next log-in may be achieved by use of LOGCPS or, because CPS never really "disconnected," by use of WRCPS with the text argument specifying the login information.

The application program, of course, may end without explicitly disconnecting from CPS. In this case the Monitor takes care of disconnecting CPS as part of the normal termination of the application program.

Interrupting CPS Activity: The experienced user of CPS is familiar with the occasional need to signal the system to change state. For instance, CPS may be doing automatic line numbering in "collect" mode and the application program has reached the point of wanting to switch to "direct" mode--for example, to request execution of the program just generated.

Another example occurs when a program in execution by CPS is in a loop, producing nothing. Alternatively, the executing program may be giving too much output because of a different type of looping. All of these needs to stop CPS are serviced by the simple procedure reference ATNCPS (Attention CPS):

```
CALL ATNCPS
```

No arguments are passed to this procedure. The action performed is dependent upon the context. If CPS is currently ready to write, the effect is the same as a terminal operator's hitting the "Break" key at a Teletype. In the case of a CPS read operation, it has the effect of the same operation being ended at the Teletype by hitting the "control-Q" (X-On) key. Finally, if the application program issues ATNCPS following a RDCPS that has timed out, it has the effect of forcing CPS to respond with a write operation, typically informing where a looping or too-slow program has been stopped by the signal.

GENERAL USE PROCEDURES

Five procedures are available for general use. Three of these, ENQ, DEQ, and DDNAME, are the work of Blair [B3]. DDNAME is a function reference whose invocation causes a two-character identifier to be returned that denotes the station address of the terminal (display or Teletype) from which the application program was invoked. Two uses of this are foreseen: programs can determine the station with which they are interacting and programs that require task-unique qualifiers for shared data sets can achieve it thus.

ENQ and DEQ provide basically the same facility as the IBM Operating System/360 ENQ and DEQ macros. That is, they allow concurrently active tasks to share access to common data sets in orderly, noninterfering fashion. Curiously, the PL/I designers did not include any equivalent facility in the language definition.

Two other procedures, ERRCODE, a function reference discussed in the next section, and DELAY are available. The DELAY call-statement is identical in purpose to the standard PL/I DELAY statement:

CALL DELAY(time)

The FIXED BIN(15,0) time argument specifies the number of seconds (the PL/I standard DELAY accepts milliseconds) that the application program execution is to be suspended. One possible use of this procedure is to allow a measured delay between consecutive display operations, where the message displayed by the first in the sequence of display operations is to be erased or overwritten by the next operation in the sequence. This allows the station operator sufficient time to read a message before it is destroyed by the next action of the application program. For over-

all system efficiency, the use of the CHAT DELAY procedure is preferable to use of the standard PL/I version. The procedure invoked by the CHAT DELAY informs the Monitor of the suspension request. This causes the Monitor to suppress scheduling overhead for the invoking application subtask for the duration of inactivity, and all time-slicing activity if no other subtasks are active. Since the Monitor is not informed of application program usage of the standard PL/I DELAY, that usage does not suppress the unneeded overhead.

EXCEPTION-CONDITION SIGNALLING

All exceptional conditions related to the CHAT Monitor-controlled facilities are signalled to the application program by raising a programmer-named condition identified as ABNORM. Each application program must define an on-unit for this condition, typically through inclusion of a begin block preceded by the condition prefix:

```
ON  CONDITION(ABNORM)
    BEGIN;
      •
      •
      •
    (action logic)
      •
      •
      •
    END;
```

Once the ABNORM condition has been raised, the application program can obtain further information on the specific condition that caused the ABNORM condition to be signalled through use of the function reference ERRCODE. Invocation of ERRCODE causes the return of one of the FIXED BIN(31,0) values shown below in parentheses:

(1) PROGRAM ERROR: indicates the application program has passed an argument with an invalid value or length attribute to one of the subroutines discussed in this chapter.

(2) SLIDE ERROR: indicates the Monitor has detected an error during execution of a slide operation that characterizes an uncorrectable slide projector malfunction.

(3) LINE ERROR: indicates the Monitor has detected an error that means

the transmission path to the display station is nonoperational.

(4) WRITE INTERLOCK: indicates a sequencing problem in the application program's use of the CPS conduit facility: both CPS and the application program are simultaneously attempting a write operation. In the case of the application program, a WRCPS or LOGCPS has been issued to cause this irterlock problem. A RDCPS or ATNCPS will break the contention.

(5) LOG-IN EXCEPTION: indicates the application program has issued one of the other CPS conduit calls when only LOGCPS is permitted.

(6) CPS DEAD: indicates a CPS conduit call issued by the application program cannot be successfully executed because CPS is not currently present in the system.

The error conditions associated with LINE ERROR and CPS DEAD are fatal ones, requiring the application program to cease further use of the corresponding facilities. In the case of the LINE ERROR condition, this means the calls associated with display usage can no longer be issued; in the case of CPS DEAD, the conduit calls are not to be used. If the application program violates this protocol, the Monitor simply stops the application program when it issues the offending call.

WRITING AN APPLICATION PROGRAM

An application program designed for the CHAT system is written with the idea that only one station exists. This means that a program intended for concurrent use from multiple displays is not burdened by the logic for handling multiple terminal operation. The Monitor, exploiting the multitasking facility of the OS/360 MVT control program, attaches the program as a separate task for each invocation of the program from a distinct display station. Thus, there are as many concurrently executing application tasks in the CHAT region as there are active terminals. Some of these tasks may be the same application program attached in multiple, while others may be entirely different application programs singly attached.

While this feature simplifies the design of an application program intended for multiple display usage, it has a restrictive effect in that an application program cannot include support for both the Teletype and a display station.

The declarations needed to define the various $-named variables and the entry points to the procedures discussed in this chapter can be included in the application program at the point in the program where the programmer has inserted the following preprocessor statement:

```
%INCLUDE (CCIDCL
          TWXDCL).
```

CCIDCL and TWXDCL are member names in SYSLIB. The confining of application program support to either a display or the Teletype is reflected in the text identifier options provided for the compile-time facility.

The CCIDCL identifier results in the generation of source text consisting of the declarations needed for display station support; this in-

cludes all procedure references and $-named variables discussed in this chapter with the exception of WRTWX and RDTWX. The TWXDCL identifier causes the generation of the declarations for WRTWX, RDTWX, ENQ, DEQ, ERRCODE, DELAY, and DDNAME. Notice that the CPS conduit facility is not accessible from a Teletype application program. (We saw no requirement for Teletype application program access of CPS, and Teletype operator access to CPS is supported by CPS itself--the operator can dial CPS directly.)

In designing an application program for multiple-display usage, the programmer has two concerns that he cannot ignore. One is the coordination of access to shared, but serial-reusable, data sets, an issue mentioned earlier in the description of ENQ and DEQ. This is a logical issue. The other concern is program reenterability. By designing an application program to be reentrant, the programmer permits the multiple attaching of a single copy of the program load module. This is a storage efficiency issue. Application programs that are not reentrant require a separate copy of the program to be loaded each time the program is attached. (Because of the presence of only the single Teletype in the CHAT System, a Teletype application need not be reentrant.)

Basically, to achieve reentrancy, the application program should be coded so that only read-only variables occupy static storage. Detailed guidelines for achieving reentrancy in PL/I are beyond the scope of this document and in fact appear to be a matter of taste (cf. Mudge [M2] and Sneeringer [S2]). The experience of using the overlay facility in conjunction with a reentrant assembler language application program is discussed by Wait [W1].

For the sake of completeness we mention a restriction and a potential-

ity which are of little significance to anticipated CHAT System applications. The restriction forbids the application program to use the multitasking option within itself. The CHAT Monitor is not implemented to allow it because of the way the Monitor Subtask Scheduler (MSS) does time-slicing and subtask scheduling--see Chapter 4. (In particular, the PL/I multitasking option is too idiosyncratic to support.) The potentiality is that concurrently active application subtasks can interact with each other, even though the CHAT Monitor does not support this. This can be achieved via common OS/360 data sets and application-program supported queue-sampling; Blair's ENQ and DEQ are useful in controlling access to the shared files. This technique is now a familiar one to the OS/360 user community; it was one of the earliest methods used to pass data between jobs in separate regions.

CHAPTER 4: REGION AND SUBTASK CONTROL

Two logically distinct CPU-scheduling services are provided by the CHAT Monitor. The first of these, time-slicing, ensures that the computational activity of the CHAT application tasks does not preempt installation CPU usage beyond a specified maximum fraction of the total time available. This is a service to the installation management who require assurance that the overall multiprogramming throughput performance not be severely degraded by CHAT region activity. During the development of the CHAT Monitor on the TUCC installation, the TUCC manager imposed the limit that not more than five percent of the total CPU time could be preempted by CHAT application computing. At TUCC, multiprogramming included, besides CHAT, six batch jobs as well as HASP, RJE, APL, and CPS (serving real teletypewriters as well as CHAT "ports"). This same limit has been carried over to the UNC installation which currently supports no interactive computing other than CHAT and only three batch jobs, along with HASP and satellite-RJE (to TUCC).

The second scheduling service provided by the CHAT Monitor is a suballocation of the region slice among the competing active application tasks. This subtask priority scheduling involves a modified round-robin scheduling technique, allowing each terminal to be given timely service.

Both scheduling services are performed by MSS (the Monitor Subtask Scheduler)--the highest priority and chief executive control task within the CHAT region. Additional responsibilities of MSS, which owns all

tasks in the CHAT region, are creation and termination of application tasks, as well as coordination of region shutdown in response either to an operator request or to certain pathological circumstances. These matters, along with the scheduling services, are described in greater detail in the following sections.

TIME-SLICING

Under OS/360 MVT, every task in the system has a limit priority and a dispatching priority, each having a numerical value in the range 0-255. The control program maintains a task dispatching queue with tasks ordered for execution according to dispatching priority. The job step task for a region is given its limit and dispatching priorities as specified by parameters in the Job Control Language (JCL) statements used to create it (and the region); this limit priority is the ceiling for all task dispatching priorities within the region. Subtasks of the job step task are given their initial priorities by parameters in the OS/360 ATTACH macros used to create them. Another OS/360 macro, CHAP, can be used to change the dispatching priority of a subtask.

In the CHAT region only one limit priority is used, and the dispatching priorities of the Monitor control tasks are all equal to it. MSS, as the job step task, is created first and thereby has the highest priority--length of time that a dispatching priority is held being the secondary determinant of dispatching order. This common dispatching priority is higher than that for the installation batch regions, but lower than that of various system tasks, e.g., HASP. Thus, the CHAT Monitor can be blocked by system supervisory activity but not by lengthy computation in the batch jobs.

The basic idea of CHAT time-slicing is to dynamically change the dispatching priority of the CHAT application subtasks, relative to the batch tasks, in accordance with the five percent formula. This alternating promotion and demotion of the application subtasks is done by MSS through use of the CHAP macro. The Monitor control tasks, because of

their supervisory role, are not themselves affected, but always remain at the same high level.

A CHAT time-slicing cycle consists of two intervals: one in which CHAT application subtasks hold high priority relative to batch, followed by one in which the relative priorities are reversed. The high-priority interval is referred to as a CHAT slice; the low-priority interval, as a CHAT low.

Computer timing, because of the presence of a digital clock, is granular--with clock resolution characterized by a smallest measurable time quantum, q. The UNC System/360 Model 75 is equipped with the standard System/360 clock, a line-frequency timer whose clocking epochs (instances when the clock value is changed) occur every one-sixtieth of a second. Thus, here q = 16 2/3 milliseconds.

The five percent formula requires that the ratio of slice to low be nq:19nq. With a sufficiently small q, the optimum value of n would be difficult to determine--requiring sufficient experience with and analysis of the running CHAT System. The grossness of the actual clock resolution, however, made the choice of n=1 an obvious one.

When no application subtasks exist in the CHAT region, or when all present are idle, awaiting an external event such as a timer interrupt or an input signal from a terminal, there is no need for time-slicing. Accordingly, the CHAT Monitor will suspend time-slicing. This period of suspended time-slicing is referred to as a CHAT slump. A run of consecutive time-slicing cycles uninterrupted by a slump is referred to as a time-slicing burst.

Initially, following initialization of the CHAT region, the CHAT state is a slump. Then upon receipt of a display station interrupt signal or of the $XEQ command from the Teletype, MSS attaches Blair's MTOC [B3] at a dispatching priority equal to the CHAT region limit priority. MSS also issues an OS/360 STIMER macro instruction specifying an interval of q (16 2/3) milliseconds. This changes the CHAT state to a slice and begins the first cycle of a time-slicing burst.

At the end of the slice interval, a timer completion exit routine specified in the STIMER macro gains control and posts an event control block (ECB), which is one of several that MSS waits on when it has no scheduled work. Under OS/360 this exit routine is an event-activated coroutine of MSS; it executes under the same task control (TCB) as MSS and has the addressing capability equivalent to an in-line subroutine of MSS without being permitted to disturb the "activation record" (Program Status Word and registers) of MSS. It does, however, have preemptive priority over the execution of any in-line code of MSS.

MSS reacts to the posted ECB by using CHAP to demote the application subtask to a dispatching priority below that of batch. Then a new STIMER is issued specifying an interval of 19q (316 2/3) milliseconds and the CHAT state is changed to a low. At the completion of the low, the exit routine again posts the ECB of MSS which in turn promotes the active subtask to the CHAT limit priority and issues an STIMER to start the slice of a new cycle.

Because the application subtasks are demoted, not suspended, during CHAT lows, they remain dispatchable at all times. This allows the OS/360 task-switching facility to dispatch any ready CHAT subtasks during periods when the other regions of the system are all in the wait state. This has

the two-fold advantage of not only allowing CHAT applications to gain faster access to the CPU resource but also shifting some CHAT activity from preemptive use of system time to system time that would otherwise be dead, thereby increasing overall installation efficiency.

In general, there can be several application subtasks in the CHAT region. For this reason, MSS keeps a chain of pointers to the TCBs for the various application subtasks that are present. This chain is used during the CHAP-scans that MSS performs to alternately promote and demote the subtasks. Indeed, there are two such chains: a high priority chain and a regular priority chain. The need for two priority groupings follows from the subtask priority scheduling scheme discussed in the next section. Here we are concerned with the chaining mechanism itself.

Within the station control block (SCB) associated with an application subtask is a field for chaining information called the subtask TCB element (STCBE). Figure 4.1 shows the format of this field. If no application subtask exists for the associated station, the STCBE is all zeros.

The first word of the STCBE holds the address of the TCB for the application subtask. This word is initialized by MSS upon attaching the subtask, and the pointer is subsequently passed as an argument on execution of the CHAP macro.

The second word of the STCBE is used for chaining. When the STCBE is on one of the chains, this word contains a pointer to the next STCBE on that chain. It contains zeros if the STCBE is last on the chain. A subtask represented on one of the chains participates in dispatching priority promotion or demotion during the CHAP-scans performed by MSS for time-slicing.

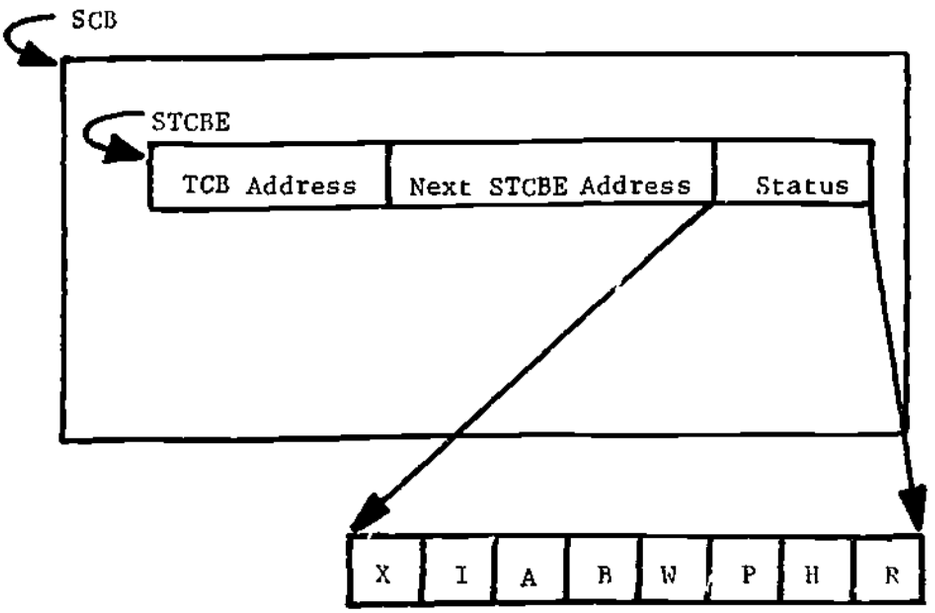


Figure 4.1 STCBE and Status Flags

The status byte of the STCBE contains information concerning the current status of the application subtask. (The X-bit is not used, while the I-bit is not pertinent to time-slicing.) The H- and R-flags are set according to which of the two chains the STCBE has been placed on by MSS. These flags are reset by MSS as it purges an STCBE from a chain. Purging is done when an application subtask is to be terminated and detached from the system or when MSS finds either the W- or P-flag set during a CHAP-scan (at the end of a slice or a low).

The W (Wait) and P (Pause) flags are set by the application subtask itself via one of the IFCSECT subroutines discussed in Chapter 2. An IFCSECT subroutine that waits for input from a CHAT-controlled source (READ, RDCPS, RDTWX, PAUSE, etc.) or for completion of a timer interval (e.g., DELAY) set one (or both) of these flags. This allows MSS to purge the idle subtasks from the chains during the next CHAP-scan. After being purged, the subtask is allowed to remain at the high dispatching priority.

The advantage of this purging is that MSS upon finding the chains empty at the end of a low can suspend time-slicing, avoiding unproductive timer interrupts and unneeded CHAP-scans. Whenever an application subtask again becomes ready, the same subroutine that originally set the W- or P-flag will report its readiness to resume execution to MSS by posting an ECB. MSS then restores the subtask to a chain and resumes time-slicing. Notice that responsiveness is enhanced by this event-activated resumption of time-slicing: in a sparsely active system like CHAT the activated subtask can gain immediate access to a slice rather than having to wait an average of 150 milliseconds if time-slicing were never suspended.

The remaining A (Abort) and B (Break) flags in the STCBE status byte are set in conjunction with two types of "CHAT-instigated" aborts to be

described in the final section of this chapter. In the case of such an abort, MSS purges the affected subtask from the chain and gives it the high dispatching priority even during a CHAT low before issuing an ABEND to abort it. The OS/360 execution of the ABEND logic involves certain critical sections that suspend all other activity within a region--even that of higher priority tasks. (The ABEND logic is executed with the dispatching priority of the task being aborted.) Thus, this feature is the single violation in CHAT of the five percent formula. Without it, the degradation of the CHAT region responsiveness was shown by experience to be intolerable during saturated loading of the installation. Subtasks were aborted during CHAT lows and MSS was unable over an extended period--hours!--to get control to execute a CHAP. This was due to the fact that MSS invoked ABEND for a subtask during a low--when the subtask had low priority. Because (1) the ABEND internal control logic then was dispatchable at the low priority of the subtask and locked out other CHAT region tasks (e.g., MSS) despite their high priority (why this is necessary remains a mystery); and (2) during heavy loading, batch regions continuously execute at higher priority, this meant the whole CHAT region was locked out until the ABEND logic could be dispatched and completed. The CHAT design was quickly altered to promote a subtask first before issuing ABEND--this corrected the problem.

SUBTASK PRIORITY SCHEDULING

The scheduling of application subtasks is intended to offer the quickest response time to those subtasks which, at the time of contention, take the least computation time to respond to input. This policy has a human factors advantage: a terminal operator making a request of an application program which involves trivial processing time can expect a quick response--and consistently so. Requests involving more prolonged processing may be somewhat delayed because of preemption by other requests arriving during their processing, but the added delay in these cases is less significant to the human operator. Freeman and Pearson [F1] show that statistically this scheduling bias results in a reduction in the variance of overall system response time--a responsiveness measure that they emphasize for batch systems as well. This basic idea of servicing small processing demands first appears generally in time-sharing systems--for which, see Wilkes [W3].

The CHAT design recognizes two priority groupings among contending (dispatchable) subtasks: a high priority group consisting of those subtasks that have become ready (e.g., by the arrival of input from a terminal) during the current time-slicing cycle, and a regular priority group consisting of those that have been continuously dispatchable for two or more consecutive cycles. These groupings result in the newly ready subtasks getting top priority with respect to other CHAT subtasks for the first cycle of their contention, and then getting a regular share of the CPU resource over the subsequent cycles that they remain continuously active.

This effectively establishes the following criterion: a quick re-

sponse capability is one in which the maximum guaranteed CPU allotment to CHAT within a cycle suffices to meet the subtask's execution-time requirements for responding. The guaranteed time is, of course, simply the slice interval--16 2/3 milliseconds.[1] This is a considerable duration on the System/360 Model 75, even considering that CHAT execution is from LCS. Adding to this whatever (non-guaranteed) system dead-time may be available (see previous section) during the low of the cycle, the ceiling for quick responses appears even more substantial.

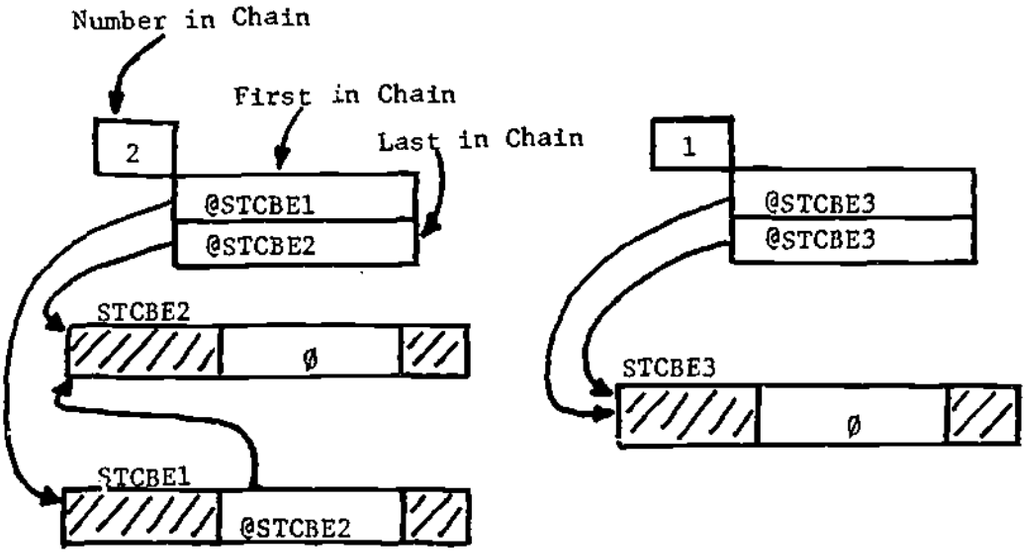
For the first cycle that a subtask is in the regular group, it has top priority within this group--being preempted only by newly activated subtasks added to the high priority group. Hence we can define a fairly quick response capability that requires some-compute requirement--namely, two slice intervals.

At the end of the second cycle of continuous activation, the subtask loses its favored status and is demoted to lowest priority in the regular group. It must then work its way up to top priority within the regular group over a period of cycles in accordance with the Monitor's round-robin sharing of the CPU resource among it and other high-compute subtask activations. The round-robin process is repeated for the duration of the high-compute activation, i.e., until it once again returns to the idle, or suspended state.

To implement this scheduling idea, MSS maintains two service chains--a regular service chain (corresponding to the R-bit previously described) and a high service chain (corresponding to the H-bit). Figure 4.2 shows the structure and a possible occupancy for these chains. Forward chaining, only, is employed because purging usually occurs only during a full scan of the chains (exceptions being primarily subtask-ends),

---

[1]Except for the first slice of a time-slicing burst which, because it begins asynchronous with the clock, averages 8 1/3 milliseconds.



(a) Regular-service Chain (b) High-service Chain

Figure 4.2 Service Chains for Active Application Tasks

while additions are made to the end of the chains.

The STCBEs on the regular chain represent subtasks that have been continuously active for more than one cycle in the current time-slicing burst. An exception is the first subtask to become active when the CHAT state has been a slump--here only the one subtask is now ready, so it is simply placed on the regular chain right away. The high chain holds those subtasks that become ready during a time-slicing cycle of an on-going burst. These can be subtasks newly attached by MSS or can be suspended subtasks signalling their readiness to resume by posting an event control block (ECB) of MSS.

The two chains are treated by MSS as follows: during the CHAP-scan at the end of a CHAT slice, those subtasks on the high chain are given a dispatching priority equal to one, while those represented on the regular chain are given a dispatching priority equal to zero. These priorities--both lower than batch region priorities--are held by the active subtasks for the duration of the CHAT low. At the end of the low, MSS begins its CHAP-scan with the high chain, then continues with the regular chain, promoting all subtasks back to the slice priority, above that of the batch regions. The high chain is emptied by placing those STCBEs found on it in their current order at the front of the regular chain.

Subtasks which are purged from a chain because they have gone inactive (waiting) are allowed to stay at the high dispatching priority. Thus, subtasks purged during the CHAP-scan at the end of a low are promoted along with the still-active subtasks, but at the end of a slice purged (suspended) subtasks are not demoted--they are only removed from the chain. Notice that long-idle subtasks will appear ahead of more recently active subtasks on the OS/360 dispatchability chain even during

a CHAT slice. This allows newly ready subtasks to gain fast access to the CPU, since the whole MSS scheduling idea is based on the management by OS/360 of its dispatching chain.

If a purged subtask becomes ready during a time-slicing cycle it is added to the end of the high chain. If this occurs during a slice, this is all that is necessary. However, during a low, MSS also issues a CHAP to demote the subtask to priority one so as not to interfere with the batch regions. So that MSS becomes immediately aware of the status change, the subtask (in the reactivated IFCSECT subroutine) posts an MSS ECB to signal readiness. On rare occasions it may happen that a subtask is suspended and then signals readiness to resume before MSS has performed the next CHAP-scan to purge it from its current chain. If the subtask appears on the regular chain, MSS will immediately, at the time of ready-signalling, purge it and then add it to the end of the high chain. If this occurs during a low, MSS also will change the subtask's priority from zero to one. Similarly, during a time-slicing cycle, newly created subtasks are added to the end of the high chain and are attached at the slice priority or at the one priority depending upon whether they arrive during a slice or a low.

To share priority among application subtasks equally, MSS performs a round-robin reordering of the regular chain at the end of a slice prior to the CHAP-scan. The current first STCBE becomes the last in the chain, while all others are moved up by one position. This new order is held through both the low and the following slice, when reordering is again performed.

Two exceptions will cause this reordering to be suspended and the current order held for an additional cycle: (1) if a subtask is detached

from the region during the current slice or (2) if the high chain is found nonempty at the end of slice. In either case, it is impossible to know how much execution time the still-ready subtasks had during the ending slice. (Recall that the slice interval is also the smallest resolvable clock quantum.)

Overall, this technique, designed to allow each subtask to gain its fair share of the CPU and to give high priority to newly activated subtasks so that quick responses are favored, seems to work very well in practice. Terminal usage experience shows the design intention to be realized satisfactorily by the implementation.

REMARKS ON SCHEDULING

Having described the details of the scheduling services provided by MSS, we consider again the necessity for such services and compare the implementation with that of other time-sharing systems.

Considering the types of applications already implemented for CHAT, it seems unlikely that the CHAT region demand on CPU usage would normally ever reach a five percent level--at least not over an appreciable period. But the Monitor cannot limit itself to just normal operation considerations, since in the real world--no matter what amount of testing has been performed--programs are apt to contain latent errors. This possibility, along with the important requirement to introduce new application programs into the CHAT production system during their development and testing, means that time-slicing has the important function of protecting the multiprogramming installation from endless loops in CHAT application programs. Only in a perfect world is there little need for time-slicing. (Notice that it is presumed that Monitor control tasks--which are unaffected by time-slicing--are free of latent endless loops. Perhaps remarkably, no such bug has ever been detected in the Monitor--even during development testing.) The same principle of protection from loops means that subtask scheduling should not be implemented as first-come-first-served with execution continuing to suspension.

It is interesting to contrast the CHAT implementation with time-sharing implementations achievable in interpretive systems such as APL or CPS. By presenting a high-level "machine" to an application program, these systems have full control of the sequencing and context of the application process through interpretive execution. This allows these

systems to parcel out CPU time by allotting each competing process in turn a fixed number of interpretively executed operations. This is not possible in CHAT since it does not present a closed machine interface to executing subtasks.

As a concluding aside, we mention the existence of IBM's Time Slicing Facility [I7], an OS/360 system generation option for time-slicing. This facility was not useful to CHAT because of its static assignment of dispatching priorities to the tasks comprising the "time-slice group." IBM's implementation assumes a static priority is established for these tasks relative to other tasks in the installation. Whenever tasks in the time-slice group have work to do (are dispatchable), they have preemptive priority over lower priority tasks in the system. Depending on what values are set for the priority of the time-slice group, there may also be tasks in the system that have preemptive priority over the time-slice group itself--potentially locking out members of the group for extended periods. Hence, CHAT would still be required to perform CHAP-scans to use this facility.

Additional characteristics of the facility, such as (1) its pure round-robin dispatching of the group members (no quick response capability) and (2) its dependence in practice on a clock of much finer granularity than the standard System/360 option, made the IBM support unattractive for use by CHAT.

OTHER EXECUTIVE FUNCTIONS

Because of its role as the job step task for the CHAT region and its ownership of all other tasks--both control and application, MSS has special requirements for executive control. Briefly, these involve matters related to application subtask creation and termination and to region shutdown.

Creation of subtasks, via the OS/360 ATTACH macro, gives MSS the right to manage the dispatchability of these subtasks (via CHAP) during time-slicing. It is also alerted by OS/360 to subtask terminations, upon which it provides reporting and clean-up services. The proctor-message discussed in Chapter 2 is originally composed by MSS through its analysis of subtask ending status. MSS also initiates subtask-terminations in response to requests to do so from the display and Teletype control tasks--matters discussed further in Chapters 5 and 6.

Region shutdown is performed when MSS detects erroneous termination of the other Monitor control tasks or upon commands to shut down from the Teletype (via MTWX). Erroneous termination of Monitor control tasks is also reported to MSS by OS/360, since MSS owns these tasks as well--see Chapter 2 ("Initialization of the Region"). The Teletype commands for shutdown are described further in Chapter 6. Region shutdown is signalled by MSS to the installstion console by means of a message saying so.

CHAPTER 5: DISPLAY I/O MANAGEMENT

This chapter describes how the CHAT Monitor manages the CHAT display cluster and coordinates I/O activity between display stations and their attached application subtasks. The Monitor control task in charge of this display activity is called the Monitor I/O Scheduler (MIOS)--a slight misnomer, since MTWX and, indeed, the conduit also do I/O.

MIOS is the largest component of the Monitor (more than double the size of the next largest) and is quite complex. The complexity is caused by (1) the inherent complications of its multiplexing services; (2) the function-richness of the display equipment, and (3) the large number of possible error conditions which need to be accommodated by its error recovery logic.

This chapter is <u>not</u> concerned with describing the primitive control of the equipment in the sense of detailing individual channel program structures or precise sequences of orders sent to the remotely-linked display cluster--even though some invention exists there. These matters are more fundamental to the equipment design than to programming options. Similarly, little will be said about the myriad possible I/O errors that can (and do) occur or the error recovery protocols invoked to handle them. Error-handling logic is challengingly tedious--with such sensibility-shattering delights as compound errors (requiring status scanning precedences), ambiguous errors, and "should-not-occur" errors--which <u>occur</u>. These matters are documented in the code listings.

The main focus here is on the control of the multiterminal/multi-application I/O activity-environment of the CHAT System. Chapter 2 elaborated on the hardware attention mechanism and link-sharing--both of which are fundamental concerns in the Monitor program design. The attention mechanism is an interesting alternative to polling--a more commonly used link-convention whereby all input signals from terminals are explicitly (and, frequently, unproductively) solicited by the computer program. Both the attention mechanism--for its control implications--and the link-sharing--because of its requirements for scheduling--are important throughout the discussion.

The basic I/O control concerns of the chapter are (1) the programming environment produced by initialization of the CHAT region; (2) the control and data structures for identifying, synchronizing and coordinating I/O in a multiple (application) request context; (3) program design features for cluster-size independence; (4) ordering of operations (by type and direction) for link-multiplexing; (5) some activity-threads involving the link-use; and (6) a general treatment of the underlying I/O control using the OS/360 EXCP interface.

Besides controlling I/O, MIOS participates in some other activities--for example, subtask control. The role of MIOS in attaching and terminating subtasks and in reporting failures--to the subtask about I/O, or to the display operator about the subtask--will also be described. Two other activities of MIOS are only briefly mentioned in this chapter--conduit access control and hard, or fatal, I/O error logging. These are described in detail in Chapters 7 and 8 which discuss the conduit and the on-line test facility more fully.

AN INITIALIZATION STEP

During CHAT-region initialization at job-initiation, a transient routine named MIOSINIT is invoked. MIOSINIT is attached instead of MIOS and after successfully performing its functions, transfers control (XCTLs) to MIOS and disappears. MIOS inherits the same OS/360-assigned Task Control Block (TCB) and becomes the Monitor control task formerly represented by MIOSINIT.

MIOSINIT obtains addressability for MIOS to the Monitor control storage and initializes the environment for I/O to proceed. This involves issuance of the OS/360 OPEN macro for the display cluster Data Control Block (DCB), initializing another OS/360 control block so that the MIOS Start I/O appendage (to be described) is known to the system, and initalizing the display complex channel adapter and multiplexer. These are all one-time requirements and need not appear in resident code.

MIOSINIT can experience problems in attempting initialization of the display complex. This initialization involves a channel program that can "hang" or repeatedly fail. Both conditions are described in Chapter 8 where remedies are also given. MIOSINIT also includes some logic for communicating with the installation console operator in the case of hard failure. Because this requirement has been obviated by the on-line test facility now available at the Teletype, it will not be described. It was useful during testing before the Teletype support was included in CHAT. The code listing describes it in detail.

When MIOSINIT successfully initializes the display equipment, it does two more things. First, it prints on the installation console: CC-7012 INITIALIZATION COMPLETE. This is useful for prompting. Second,

it initializes a linkage required for MIOS to be signalled when the System/360 Attention status occurs in the System/360 channel status word (CSW) associated with the physical channel port for the CC-7012 channel adapter.

The Attention status plays an important role in the System/360. It allows the channel adapter to signal across the System/360 channel even when no channel program is currently active. It does, however, introduce added complexity in the system control program in order for the I/O Supervisor (IOS) to identify to whom the status should be signalled. When a channel program is active this is no problem, since a chain of OS/360 control blocks links the Unit Control Block (UCB) to the using program, and the UCB is itself linked by sysgenned location to the physical port. Without the chain the task is more difficult.

OS/360 offers a sysgen option, whereby an Attention-handling routine can be specified and its address tied to a field in the UCB. At Attention-time this routine is then invoked (regardless of whether a channel program is or is not active).

CHAT has such an Attention-handling routine, whose invocation simply results in its posting an Event Control Block (ECB) belonging to MIOS. This CHAT Attention-handling routine was coded by Blair who also describes the sysgen protocol for including it [B3].

To inform this routine where the MIOS ECB is, MIOSINIT issues Blair's SVC 239 [B3] passing the address of the ECB. This links MIOS to the Attention-handler. For added generality, the protocol also includes passing the address of the CC-7012 UCB (via the ECB). The Attention-handler associates the UCB with the ECB in its control storage so that the Attention-handler can be shared across UCBs.

After completing this last step, MIOSINIT transfers control to MIOS.

WAITING, LINKING, AND QUEUEING

Most of the time MIOS does nothing at all but wait for one of the events it handles to occur. If this were not the case, MIOS would represent too much overhead and CHAT would be inefficient. When MIOS is activated by one of its events, it has three immediate problems: (1) What happened? (2) Who did it? (3) What can be done about it?

The first problem and, in part, the second are solved by the wait-protocol that MIOS follows. Figure 5.1 shows the list of ECB addresses (and their meanings) that MIOS waits upon (using the OS/360 WAIT macro) when it has nothing else to do. The list is shown in the order that MIOS scans it when one of the listed ECBs is posted. Apart from the urgency associated with a shutdown request, the ordering is not overly significant.

Shutdown, $E72 (a request from MTWX to Enable the CC-72 multiplexer, discussed further in Chapter 6), and the Attention signal are not single-station-associated activities. Who posted them and what is to be done are clear; the details will be deferred to later sections.

An I/O completion presents the problem of determining the station (if any) with which the current I/O is associated. Once this is determined, MIOS must decide which I/O operation to initiate next.

The other events associated with the ECBs shown in Figure 5.1 are also station-associated. However, part of their posting-protocol involves passing the address of the SCB (Station Control Block) to which the event applies. This is logically redundant since the ECBs themselves are located in their associated SCBs; the protocol is followed for efficiency reasons.

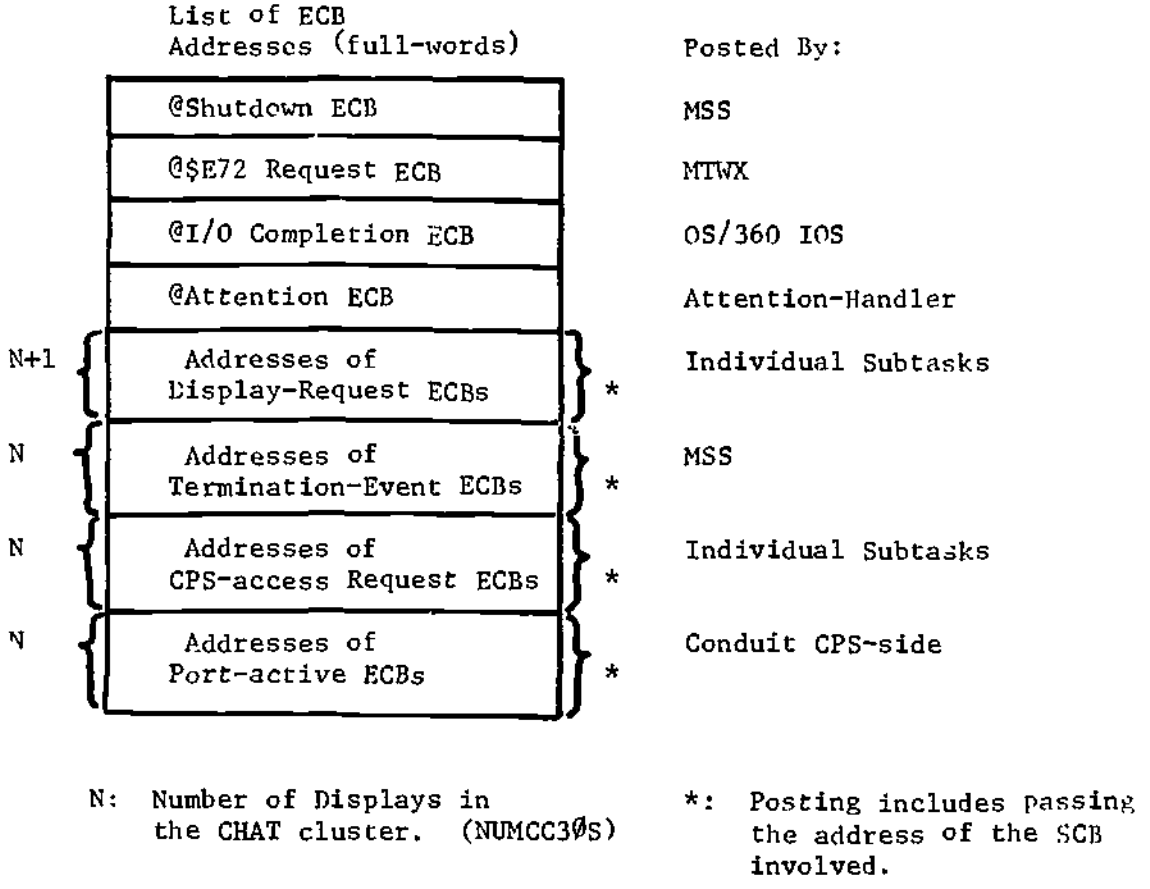


Figure 5.1 MIOS ECBs and Their Posting

While station-association in these cases is no problem, honoring the request associated with the ECB posting might be. Display requests can arrive while I/O is in progress for another station, or the request (e.g., to read) might require an Attention-signal (display activity) first before anything can be done. A termination event, on the other hand, may require MIOS to send a proctor message (see Chapter 2) to the display; this, too, is affected by current I/O activity. The other two event classes involve the conduit and are described in Chapter 7.

Figure 5.2 shows how both problems are solved by MIOS; for brevity, we assume only three display stations. As shown in the figure, MIOS holds I/O requests in queues; one queue exists for each type of operation. (Exceptions involve multiplexer-only activity.) Indexes are used to allow small queue elements (one byte per index) and to locate the addresses of their associated SCBs via indexing of the list of SCB addresses (SCB@LIST) residing in MIOS control storage. (STATINDX is described in a later section.)

When an I/O request occurs, MIOS obtains the index from the SCB and places it at the end of the queue associated with the request type. The counts shown are also updated. Whenever MIOS initiates an I/O operation, it stores a code defining the type. (There are more types than queues.) Upon completion of the I/O, MIOS branches to the routine that handles the type; and this routine, by knowing which queue is involved, can locate the SCB involved--the first index in the queue is used. The index is then taken off the queue and the counts updated. Queue-handling subroutines are invoked for putting on and taking off--the latter returning the SCB address. When I/O scheduling can be performed after I/O completions, the counts are used to determine quickly what activity to start, if any. The

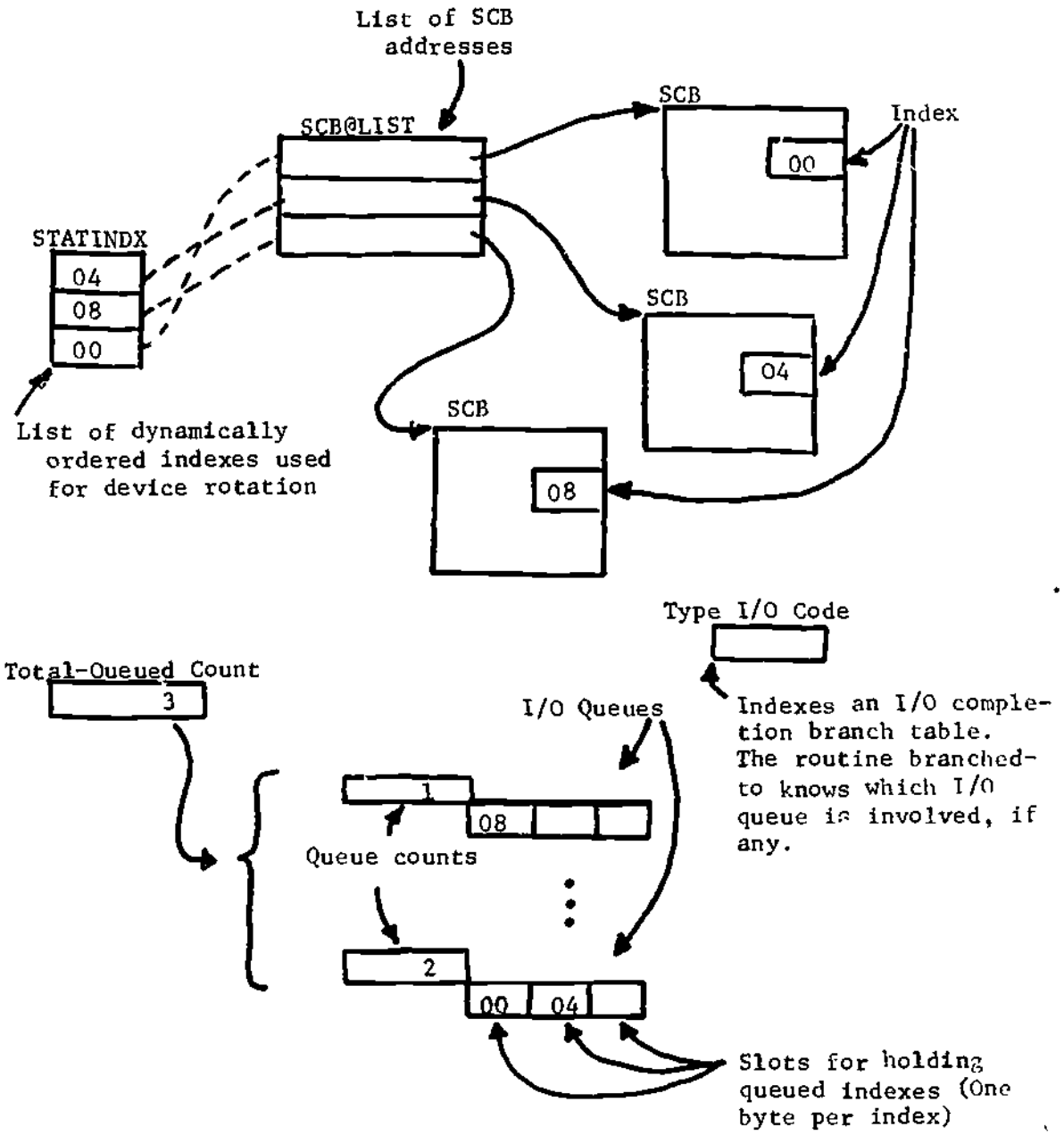


Figure 5.2 SCB Indexing and Queues

scheduling involves a priority scan of the queues to be described. Figure 5.2 shows a possible occupancy of the queues in which all three displays are active.

Two exceptions to immediate queueing of requests are possible. If the request is READ, READLP, or PAUSE (none of which can be forced but require operator activity) and no interrupt is yet pending, the request is queued when the interrupt later arrives. On the other hand, if one of these requests occurs after the event that would satisfy it has occurred, the completion is posted immediately to the subtask. (Recall from Chapter 3 that a lightpen interrupt satisfies a READ as well as READLP and PAUSE.)

INSENSITIVITY TO NUMBER OF DISPLAYS

MIOS is insensitive to the actual number of displays in the CHAT System configuration, allowing any number from 1-32 (the multiplexer maximum). To increase the current number defined, the CHAT system programmer need only code a CHAT-defined macro (SCB) for each new display, specifying a few display parameters, and change a single equate card (NUMCC3ØS) that specifies the new number. By then reassembling all of the CHAT Monitor components, the process is complete. (More details on the SCB macro are given in Chapter 9.)

To achieve this insensitivity, MIOS depends exclusively on the NUMCC3ØS parameter in all of its code. This means that list-lengths and queue-sizes are everywhere determined by this parameter. (This is complicated--particularly because of cross-references between Monitor control storage and the SCBs--by the need also to initialize lists of addresses at assembly-time.) Similarly, all loop control for scanning and searching through the SCBs is parameterized. Where display-dependent values are required, such as for display addressing, these values are fetched by MIOS from their standard locations in the SCB currently in use--the SCB macros assembled them there.

This idea is illustrated in Figure 5.1, where N denotes the current value. The N+1 shown is not a violation of this idea, but accommodates the Teletype-attached application subtask (described in Chapter 8) that "seizes" a display and makes display I/O requests. The Teletype is not accommodated in the other lists because termination events are diverted to MTWX, while CPS-access is not defined for this usage. (It isn't needed.)

This insensitivity through parameterization is total in MIOS and,

indeed, is total throughout the CHAT Monitor, with the exception of the conduit. Chapter 7 describes some minor limitations in its design and some sysgen needs for increasing Teletype UCBs that it uses. Even the conduit, however, is parameterized except where unavoidable. The effect of this implementation is that as the CHAT System grows its configuration can be redefined by anyone with even the most cursory knowledge of its software support. It does not require the presence of its designer.

SCHEDULING RULE

In Chapter 2, we described the general rule that MIOS favors input into the CHAT region over output from the applications to the displays. Now we elaborate upon this rule and discuss the detailed implementation.

Table 5.1 shows the scanning order for the I/O queues when a new operation is to be scheduled. Lightpen operations (retrieval of lightpen coordinates or lightpenned character) are performed automatically by MIOS at the time of Attention-signalling which is asynchronous with the application subtasks' explicit requests for lightpen information. Attention-handling is always exhausted through this step for all active stations before other operations are performed. Keyboard enabling is scheduled when a subtask issues a request for interrupt notification (READ, READLP, PAUSE) and MIOS notices that the display station's keyboard is locked. It is given high priority so the station operator can begin to use his keyboard. Input operations requested via READ are also given good priority to provide quick mechanical response to the operator's hitting of the INT key after typing. The last high priority is given to extra-interrupt signals (those received when a previously received interrupt has not been serviced by the application subtask) which MIOS detects from the display. This servicing involves reading the five characters preceding the current location of the cursor and either terminating the subtask if ABORT (or abort) is read or placing a "?" symbol at the cursor location if ABORT is not read. This assures the operator that CHAT is alive and active even though the application may not be.

The next check MIOS makes is to examine an interrupt-pending flag

Priority	Queue
1	Lightpen Operations
2	Enable Keyboard Operations
3	Input to Subtask Operations
4	Extra-Interrupt Servicing
5	Interrupt-Pending Flag*
6	Slide Actions
7	Output to Display Operations

*Not a queue

Table 5-1 I/O Queue-Scanning Order

before scanning the low priority queues. This flag is turned ON only after a low-priority operation has previously completed. These operations always entail a channel programming step that reads in the Short Status (SS) described in Chapter 2. Within the SS is a flag telling whether the CC-72 multiplexer has some interrupt status buffered in its Station Interrupt Status (SIS) register. When MIOS detects this flag at I/O completion time it sets the interrupt-pending flag for use in its next scheduling scan. At that time, it will defer serving the two low priority queues and poll the SIS register just as it does when an Attention is signalled. Notice that this rule gives good service to all stations even if one subtask is doing an enormous amount of repeated output (perhaps because of a loop). After each output message, MIOS checks for any input requests to be serviced and satisfies all those found before continuing with more output. In the case where all subtasks are doing output only, no subtask can lock out the others. Each output request (maximum: 810 characters transmitted) involves CPU time to invoke the DISPLAY procedure (see Chapter 3); the round-robin priority scheduling of subtasks described in Chapter 4 ensures that each subtask is given CPU time to issue its output request.

After MIOS has exhausted all queued requests, it enables both the CC-72 and the CC-7012 for a new Attention status to occur and waits for some new event to happen.

SERVICING ATTENTIONS

When an attention-signal is received, or if the interrupt-pending flag is ON, MIOS schedules an operation to read the SIS register of the CC-72. This was previously described in Chapter 2 as involving a 6-character encoding representing both the Short Status (SS) and the 32 station-interrupt status (SIS) flags.

Upon receipt of the SIS characters, MIOS uses the station index list (STATINDX) shown in Figure 5.2 to determine the order of checking the interrupt status. MIOS picks up the first index in this list to choose the starting point in the SCB@LIST. When MIOS is done using the SCB thusly chosen to see if an interrupt has occurred at that station, it picks up the SCB address indexed by the next index in STATINDX. This continues until the STATINDX list is exhausted (controlled by NUMCC3ØS loop control). At this time, MIOS reorders STATINDX (also controlled by NUMCC3ØS) such that the indexes are all moved up by one in this list, except for that index that had been first. This one is now put at the end of STATINDX so that a round-robin device rotation is effected. At the next Attention, this new order will be used and STATINDX again reordered.

The reason for device rotation is to avoid any possible display station favoritism that might become visible at the remote site. When concurrent interrupts are received and handled, the first one found is the first one placed in an operation queue. A fixed scan might eventually become known and operators then might vie for the favored displays. This has been countered by dynamic reordering of STATINDX. (Its significance has been greatly diminished by the increase in bandwidth resulting from the move of CHAT from TUCC to UNC.)

When an SCB address is picked up, MIOS performs a test using a byte-offset and a bit-mask obtained from standard locations in the SCB. Each SCB has unique values assembled in the standard byte-offset and bit-mask field locations. The byte-offset indexes the SIS character string, while the bit-mask corresponds to the interrupt-status bit in the indexed SIS byte associated with the current SCB. This allows MIOS to check if the associated station has an interrupt pending without requiring code sensitivity in MIOS to the status checking.

If a match is found, MIOS does one of three things. If no subtask is attached for the interrupting station it signals MSS to attach one. If an interrupt is already pending for this station, it places the station index in the extra-interrupt servicing queue. Otherwise, it places it in the lightpen operation queue.

Completion of the subsequent lightpen operation (getting the coordinates from the CC-301 lightpen register) defines the type of interrupt. A bit in the received coordinates is set before being transmitted by the remote display according to whether the coordinates were valid (search character present). If the search character was not present at the time of the interrupt, the remote display will set the bit to the invalid state. This permits MIOS to queue an application subtask's READ request (if one). If the coordinates are valid, MIOS completes the lightpen operation by scheduling an operation to get the lightpenned character. These lightpen operations proceed even when no application request is present and the results are saved in the SCB for later pick-up when the request occurs.

It is possible that interrupts may be received when the subtask wants to do output. In these cases, the output will be scheduled after

MIOS has finished the lightpen protocol (and discarded the results). Details are to be found in the code listing.

I/O INITIATION AND COMPLETION

This section describes I/O activities in general. Only the gross mechanisms and issues are discussed. Details of the individual operations may be found in the code.

Channel Programming: Once an operation has been chosen for initiation by the scheduling rule, MIOS chooses the appropriate channel program. There are ten of these, and most are multistep channel operations of some complexity. Besides controlling the direction of transmission over the System/360 they send a large number of display station and multiplexer control characters to control the transmission environment there for the purposes of sending and receiving data.

Most channel programs are skeletal, requiring different types of filling in of variables to meet current usage needs. Once filled in, the channel program is ready for use; MIOS uses the OS/360 EXCP interface. This filling in and initiation involves a change in the protection status of MIOS: they are executed when MIOS is in keyzero state. The reason for this is that the channel programs and the buffer MIOS uses are in the MIOS program module storage and require keyzero privilege for storage modification. By issuing the EXCP in keyzero, the channel program can subsequently read into the protected buffer. Once issuing the EXCP, MIOS leaves keyzero state and reverts to its normal TCB key. This placement in the program module is to give added protection against storage violations damaging them--I/O is complex enough without confounding errors.

MIOS also defines a Start I/O appendage. This is invoked by OS/360 IOS after the EXCP but prior to executing the System/360 Start I/O

instruction. The sole purpose of this appendage is to change the channel program starting point in the System/360 Channel Address Word (CAW), in the case where the operation involves a restart for error recovery. Here, the channel program origin is set to the restart location (obtained from the standard restart field in the IOB) so that certain origin and offset information needed for MIOS hard error handling is preserved. Then the appendage exits to IOS for further processing.

When output is being performed, MIOS first moves data from the application subtask area into the buffer before issuing the EXCP. Both the location and size of the application area are obtained from the SCB as the IFCSECT (discussed in Chapter 2) routine posting the request has initialized it, using the PL/I parameter lists (dope vectors). The data is translated after being placed in the buffer. The channel program may also send out various display control characters present in fields of the SCB as initialized by MIOS.

Completion and Posting: Completion of I/O results in the reverse process. On input the data in the buffer is translated to EBCDIC from ASCII and moved into the subtask's area. The channel programs, however, are subject to a multitude of errors. Hence, prior to completion, MIOS may have invoked error recovery procedures to correct transient errors. Sometimes errors cannot be recovered from and MIOS must regard them as hard errors.

Posting to the requesting IFCSECT routine may be of two types: normal and abnormal. The first causes the requesting routine to return normally to the application, while the second causes it to raise the ABNORM condition described in Chapter 3. A third posting by MIOS is defined for a

subtask request that violates the rule not to continue to use a line for which LINE DEAD has been signalled. This posting is done in lieu of the EXCP and causes the requesting IFCSECT routine to invoke the PL/I stop routine for the subtask.

HARD ERROR HANDLING

Hard errors are defined as occurring when the error recovery procedures reach a maximum failure count on a single operation. Chapter 8 describes the details of the error logging by MIOS and how an error record can be accessed from the Teletype. Here we describe the other activities performed by MIOS at the time of hard error detection.

If the error is confined to a single display station, the request is first dequeued and the subtask is notified. In the case of a slide projector failure this is all that is done (see Chapter 3) and the subtask may even continue using the display. In the case of a display failure, MIOS prints the following message on the installation console: I/O ERROR ON CCI STATION. The subtask is not permitted further access to it.

In the case of a multiplexer or channel adapter error the procedure is more complex, since now all subtasks are affected. MIOS purges all of its request queues and notifies all waiting subtasks. Other subtasks are notified as they issue requests. In addition, MIOS prints the following message on the installation console: I/O ERROR ON CCI COMPLEX. It then goes to wait for a new event to occur.

SHUTDOWN

MIOS executes its shutdown protocol in response to a request from MSS (the Monitor control task controlling the CHAT region and all tasks in it). This may be either because MSS has detected abnormal termination of MTWX (the Monitor Teletype control task) or because MTWX has relayed to MSS the request from the Teletype (see Chapter 6).

The first step MIOS performs is to use its SCB@LIST (Figure 5.2) to scan for any active subtasks. Whenever one is found (a TCB address is present in the SCB), MIOS posts a request to MSS (using an ECB in the SCB) to terminate it abnormally. This is what shutdown implies--stop everything immediately.

In addition, MIOS zeroes out its own TCB address located in the conduit. This requirement is discussed in Chapter 7. Finally, MIOS issues Blair's SVC 239 again, this time to pass the location of an MIOS control storage field holding the address of the CC-7012 UCB. This is to stop the Attention-handler from continuing to post future Attention status to MIOS. MIOS then exits.

CHAPTER 6: TELETYPE CONSOLE SUPPORT

At the time of the original CAI Project meetings to set design objectives for CHAT, the role of the Teletype was fuzzy. The Department already owned a Teletype and a connection port at the System/360 which were to be inherited by the CHAT System. Only some vague notions that a CAI proctor terminal would be desirable kindled interest.

One such notion was that hard copy of CAI statistics obtained interactively by the proctor might be useful. This implied a need for the CHAT Monitor to include Teletype I/O support and an I/O interface for application programs that served proctor inquiries. CHAT meets these requirements by the PL/I I/O interface support described in Chapter 3 and by the inclusion of a Teletype I/O control task named MTWX within the CHAT Monitor.

In practice, the Teletype interface has not yet been applied for CAI inquiry. Instead, programs producing CAI statistics use the high performance system printer and the installation batch facilities rather than CHAT. Blair designed ENQ and DEQ [B3] (see also Chapter 3) to work system-wide for PL/I (as OS/360 provides at the assembler level) so that batch jobs can share files with CHAT application subtasks while the latter are active. This power, combined with the geographical nearness of the CPU installation to the proctor, has lessened the interest in Teletype-based CAI inquiry.

Another role for the Teletype has come to the fore, stemming from

its convenience for allowing CHAT region control from the CAI site directly and interactively with the CHAT Monitor. Hence the emphasis in this thesis is on the Teletype as a CHAT submonitor console. Functions such as selectively terminating CHAT application subtasks and shutting down the CHAT region can be done from the Teletype by commands to MTWX. CHAT initiation still requires calling the installation's human console-operator. (During the TUCC era, the proctor could also initiate CHAT from the Teletype by dialing the TUCC support for remote job entry. This was useful during development but not in production since it means running under control of HASP.)

The remainder of this chapter discusses detailed usage of the Teletype, the command set, Monitor messages to the Teletype, and some details of the implementation and structure of the Teletype support. Chapter 8 describes a CHAT facility for on-line testing that is built on the Teletype I/O application interface; it makes the Teletype an even better console for CHAT.

DETAILS ON TELETYPE USAGE

In the System Overview (Chapter 2) we covered the general usage of the Teletype; here we add some details.

Modes: Because the Teletype can be used to acceas both an application subtask and the Monitor, MTWX needs to distinguish between the two types of access. When no application subtask is attached, this is simple: any input must be a command and MTWX is in command mode. When an application subtask is present, MTWX is normally in application mode. In this mode MTWX will pass all input to the application subtask unless it detects a $-prefixed command from the set described in the next section; this will be treated as input to MTWX. (Input such as $1.98 would be given to the application program.) After execution, MTWX remains in application mode. MTWX will change from application mode to command mode if the Teletype operator uses the Break key during output of application data (stopping it in mid-line possibly) or when the application program is executing something other than a Teletype I/O statement. This latter condition holds when the Teletype carrier is at the leftmost position of a new line and the "?" character is not displayed. Forced into command mode, MTWX again requires a command. After receiving and executing one, MTWX will restore application I/O and return to application mode--unless the command intentionally destroyed the subtask or the region. Notice that application mode allows interleaving of commands and application data.

Break has no functional meaning in command mode although it can disturb MTWX I/O. In such cases error recovery logic overcomes the diaturb-

ware--e.g., MTWX retransmits a disrupted output message. Occasionally, MTWX will itself issue a Break signal in order to clear certain line conditions. Use of Break in either direction requires that the operator hit the Break-release to unlock the keyboard.

Paper Tape: MTWX supports paper tape usage. When Break is used to interrupt application mode, MTWX suppresses its normal paper tape activation to allow keyed input. On return to application mode, MTWX resumes paper tape activation.

Ending a Message: The philosophy of the CHAT support is that the Teletype is fundamentally a typewriter (despite its lethargic button keyboard). Thus, when the operator has typed all he needs to on a line, he need only hit the X-Off button to signify completion. The Monitor provides the functions of carrier-return and line-feed. This design bias stems from the preference of the author for the IBM 2741 "Return" key (which provides all three functions by hardware) over the Teletype 3-button approach, as well as his distaste for software support that requires the operator to hit the three buttons before it accepts a line--cf. TUCC's RJE. The CHAT-supported carrier-return/line-feed also provides visible assurance that CHAT, at least, is there. CHAT allows the 3-button approach (which can be useful in producing paper tape to be locally printed) but strips out the carrier-return/line-feed, here as well as anywhere else in the transmission, prior to delivering the input to the application program. It also sends its usual carrier-return/line-feed in acknowledgment.

Similarly, the Monitor ends an application program write (see WRTWX in Chapter 3) by also sending carrier-return/line-feed.

MONITOR COMMANDS

The following $-prefixed commands are supported by MTWX for CHAT subsystem control. Syntactic and contextual abuses of the commands are discussed in the following section.

$E72 This command (Enable CC-72) is included in the command set to counter a deficiency in the design of the display hardware complex. The CC-72 display multiplexer is enabled to send an inquiry (attention) control character, signifying readiness to transmit from a display station, only through the gracious services of the S/360-resident Monitor (MIOS): MIOS sends an explicit enable order to the multiplexer. Once enabled, the CC-72 disables itself at the time when it sends the inquiry signal and awaits an interrupt-register poll by the signalled program. (Of course, the Monitor can also explicitly disable the CC-72 at will to avoid unnecessary transmit-contention.) Unfortunately, synchronism-loss because of line noise can prevent the Monitor from detecting the inquiry; if the Monitor has no need to communicate to the CC-72 (say, no application subtasks are present or else all present subtasks are awaiting non-timed input) the Monitor will remain ignorant of the readiness of the remote displays. This is because no "enable-switch" exists at the remote CC-72 allowing the fully conscious human operator to intercede and to re-enable the equipment manually.

The command causes MTWX to post an MIOS event control block alerting MIOS to execute a CC-72-enabling channel program. Command execution does not guarantee that the CC-72 will become enabled,

since other hardware errors may be present.

$XEQ <load module name> This command (load and execute) causes MTWX to invoke a Teletype application program (if one is not already attached). One blank (or space) is required between the command and the <load module name> which must have standard OS/360 format.

$ABORT This command causes MTWX to request OS/360 (via MSS) to abnormally terminate the currently attached Teletype application program. It is useful for terminating a looping application program or avoiding lengthier application-defined sign-off protocols.

$ABORTnn This command allows abnormal termination of an application program attached to a display station identified by the nn identifier, which corresponds to the display station address on the CC-72 multiplexer--also prominently displayed on the door of the room in which the appropriate display resides. Evidence of the subtask's complete removal from the system is displayed at the display station by a proctor message.

$SHUTDOWN This command causes MTWX to disconnect the Teletype, to request termination of any Teletype application subtask present, and to notify MSS of the shutdown request, while terminating its (MTWX) own presence in the system. MSS in turn notifies MIOS of the shutdown and, after termination of all display application subtasks and MIOS, MSS displays a message on the CPU system console and exits. This removes CHAT from the installation.

$RESTART This command is currently identical to $SHUTDOWN (except for certain future-relevant bit-settings). The idea is that this command will someday [B3] allow a shutdown followed by automatic refresh of the CHAT region. Currently, refresh involves intervention by the installation CPU console operator, who simply performs the standard console-initiation of CHAT.

$NULL This is the null command. It does nothing except to cause MTWX to change from command mode to application mode when an application program is present (one can hit the Break key accidentally). Otherwise, it is simply an acknowledged "No-Op."

MESSAGES SENT TO THE TELETYPE OPERATOR

This section describes all messages which the Teletype operator can receive from the CHAT Monitor.

MTWX HERE...ENTER A COMMAND This message is sent:

(a) when the operator successfully dials into the CHAT region,

(b) immediately following the COMMAND EXECUTED message (below) when an application subtask is not present,

(c) following the receipt by CHAT of the Break signal when an application subtask is active, or

(d) when an attached Teletype application subtask has just terminated.

It signifies that MTWX is in command mode.

COMMAND EXECUTED This is sent to acknowledge that the last $-command was executed. If an application subtask is present, it also signifies the escape from command mode to application mode. This message is not sent, however, following $SHUTDOWN or $RESTART since the resulting disconnection of the Teletype is visible-enough evidence that the command was recognized and honored.

NOT A LEGAL COMMAND...RETRY This message is sent for a variety of reasons:

(a) a syntactically incorrect (misspelled or no $-sign) command has been received in command mode,

(b) $ABORT has been received but no application subtask is present,

(c) $XEQ has been received with too long (>8 characters) a load module name specified,

(d) $ABORTnn has been received but the nn does not correspond to any known display station in the system, or

(e) $ABORTnn has been received and the nn does correspond to a known display, but it is the one currently "seized" by the Teletype application subtask (OLTEST). (The notion of seizure is described in Chapter 8. A $ABORT command would succeed here--getting rid of OLTEST and "releasing" display nn at the same time.)

SUBTASK ALREADY PRESENT This is sent when MTWX receives a syntactically correct $XEQ command while a Teletype application subtask is currently present in the system. Because of system delays in abnormally terminating a subtask, this message may be received even though MTWX has just previously acknowledged honoring a $ABORT command. (MTWX acknowledges as soon as its part in instigating the process is complete; evidence of the subtask's presence remains visible to MTWX until OS/360 and MSS complete its removal.)

SUBTASK ENDED: <code> This message was discussed in Chapter 2 (cf. "proctor messages" in the "Terminal Usage" section) where <code> signifies system (OS/360) or user (application) ABEND codes or user return codes. It is not sent after the Teletype operator has used $ABORT; it is presumed that the operator remembers that he caused the termination. (However, $ABORTnn does result in a proctor message being displayed at the affected display station to pro-

tect against devilish or malicious tampering from a Teletype with display stations. It also confirms removal-completion there.) Note: After MTWX has acknowledged a $XEQ, it may happen that this message will immediately appear with an "Snnn" code, signifying "load module not found" or "load room (CHAT free core) not available." These errors are found by OS/360, not by MTWX.

INPUT TOO LONG This message is sent because a message from the Teletype appears too long to the Monitor message editing logic (an "IOS appendage"). It is either the Teletype operator's fault or due to a communication line error (sometimes indistinguishable). In command mode the total transmitted message length should not exceed twenty characters (including controls such as X-Off and the CHAT-defined character-delete); in application mode the total transmitted message length should not exceed ninety characters (including controls) or (after editing) eighty, without controls. (An application program is constrained by CHAT to a maximum input length per read of eighty.) The MTWX channel program allows one to transmit a message of infinite length, but when one finishes (!), this message will appear. (The overflow is read into a single storage byte via a channel program READ/TIC loop.)

LINE NOISE...REKEY This message is sent when the message editing logic detects via S/360 TRT-scanning of the input that a parity error exists in the input data (the hardware does no parity checking for the Teletype). If this message is frequently seen during a connection, the operator should try disconnecting and redialing. If it

is persistently frequent across connections, inform the local telephone company communications specialist.

STRUCTURE OF THE TELETYPE SUPPORT

The Teletype support involves the cooperation of a number of Monitor components. The main one is MTWX which is one of the three Monitor control tasks. MTWX initiates Teletype I/O using the OS/360 EXCP (Execute Channel Program) interface with dynamically modified CHAT-defined channel program skeletons. MTWX controls the overall sequencing of I/O operations in accordance with:

- application program requests,
- current activity at the Teletype,
- requests to shut down, (the Monitor control task controlling subtask scheduling)
- reports from MSS of Teletype application terminations, and
- I/O "hard-error" reports.

All $-command interpretation occurs in MTWX although MIOS (the Monitor control task controlling the displays) and MSS may be invoked to help.

Like MIOS, MTWX interfaces with an application program through subroutines included in IFCSECT described in Chapter 2. Recall that these subroutines synchronize application activity with I/O completions and request I/O by post-wait protocols. I/O type and application parameters (I/O area, length) are passed to MTWX and MIOS via fields in the appropriate station control block (SCB).

The SCB for the Teletype contains the same standard fields as a display SCB but also has an extension for Teletype-only requirements. The standard portion of the Teletype SCB allows MSS to be unconcerned with the identity (MIOS or MTWX) of the requester of standard Monitor services related to application subtask scheduling (attaching, aborting, detaching) which are independent of whether the application is servicing

the Teletype or a display. In other cases there is a need to distinguish between the two types of SCB--an example being to prevent an operator loading a display application program from the Teletype or vice versa. Hence, one of the standard SCB fields has a flag denoting the terminal type.

MTWX has help on the I/O Supervisor (IOS) side of I/O handling. The OS/360 EXCP interface allows the programmer to specify up to five appendages (for details, see the IBM OS/360 System Programmer's Guide). Two CHAT-defined appendages (called CEAPP and ABAPP) have been included for the Teletype support. These are invoked by IOS as subroutines at channel end/device end interrupt-handling time. IOS chooses one or the other of the IOS appendages depending on whether the channel end/device end status is (ABAPP--Abnormal End Appendage) or is not (CEAPP--Channel End Appendage) accompanied by unit check status. Unit check means an I/O error.

The IOS appendages provide a number of functions:

- input editing including control character stripping,
- character deletion,
- line cancel,
- translation to EBCDIC,
- parity checking, and
- length checking.

Error recovery is performed in the appendages without notifying MTWX unless a retry threshold is reached for a single channel program. In this case, the appendage sets a fatal, or hard error code that is posted to MTWX.

Appendages return to IOS via one of three exits (actually four are

allowed but one is not used here) which determine whether IOS will

(1) post an I/O completion to MTWX using a code passed by the appendage,

(2) ignore the interrupt (by not posting), or

(3) restart I/O according to an appendage-set restart point.

CEAPP and ABAPP use exit (1) to inform MTWX that a normal completion or hard error has occurred. Exit (2) involves Halt I/O activity initiated by MTWX but whose completion is not of interest. Exit (3) is the main advantage in appendage-processing. It gives quick handling of error retries as well as quick dispatching of queued I/O requests--a matter relating to special use of the "Prepare" channel command within the Monitor for Teletype monitoring. It allows continuous monitoring of Teletype activity when no other I/O activity is ready to be scheduled. Details are documented in the code.

MTWX has some logic also for accommodating the display-seizure function described in Chapter 8. This is minor--involving primarily cleaning up (releasing the display) if the subtask that seized the display has abnormally terminated or is requested to be (via $ABORT) by the Teletype operator. This function of seizure has also enlarged the Teletype SCB beyond the need to preserve standard fields; seizure permits concurrent usage of display and Teletype, so a larger SCB is needed that permits both activities to use it.

CHAPTER 7: THE INTERREGIONAL CONDUIT

The interregional conduit is a CHAT feature designed for a special need of Mudge's DIAL [M1], an application program running under the CHAT Monitor. This need was for access to the string-handling and other capabilities of the IBM Conversational Programming System (CPS) [I1].

An early question was: How? Storage limitations ruled out the notion of including whatever portion of CPS was needed within DIAL or elsewhere in the CHAT region. Such inclusion in any case would have been particularly wasteful, since a full copy of CPS resided (at TUCC) in another region as a nonterminating job. Thus, the only answer was to somehow use the already-resident copy of CPS. This imposed the requirement on the CHAT Monitor to provide an interface between the application subtask and CPS. Three critical problems were:

1. That copy of CPS belonged to the TUCC community and it was clear that TUCC management would (rightly!) not tolerate abuse of that valuable property.
2. The CPS program was very big, very complex, and very poorly documented. In fact, the internal specifications manual was virtually worthless and the code listings were mostly empty of comments.
3. No facilities exist in OS/360 for interaction between separate regions. Indeed, a central concept of OS/360 is to completely insulate one region from another because of protection concerns.

The author chose to tie to CPS by means of one interface it already presented to the world--namely, its terminal interface. This has the following advantages:

1. This interface is an external one that is well documented [I1].
2. It is an interface using programmed support (I/O channel programs) in CPS that is stable and unlikely to change from release-to-release of that product.
3. The I/O support in the CPS code, while atrociously commented, could be understood with study because the functions provided were understood.
4. CHAT's interface to its application programs could offer the well-known and powerful functions familiar to the application programmer through his own CPS terminal use.

CPS offers programming support for the IBM 2741, the IBM 1050, and the Teletype. The Teletype interface was selected. This has no impact on CHAT's application interface (see "CPS ACCESS" in Chapter 3) since terminal-dependencies such as line codes and control characters are not visible to application programs. On the other hand, this selection was beneficial to the author whose experience already included OS/360 BTAM programming of the Teletype and who had ready access to a Teletype. This aided in learning CPS's I/O logic by exercising it and doing software "snapshot" probing. This was handy in getting through some of the really mysterious code in CPS and also in debugging the CHAT conduit. It was also an opportunity to learn functional support (better channel programs, richer interactive orientation) superior to that offered in BTAM.

The CHAT MTWX Teletype control and channel programs borrow concepts

obtained from this study of CPS--in particular, the use of the Prepare command for constant line-monitoring (along with Halt I/O to allow scheduled operations) and the avoidance of unnecessary "time-outs" (by using Inhibit instead of the Read channel control word).

The consequence of this design of communicating with CPS via its Teletype I/O interface is that not a single change has had to be made to the CPS program. Only additional data definition (DD) cards (see Blair [B3]) in the JCL initiating CPS are necessary. These specify the existence of additional Teletype address ports on the System/360 channel--although none such exist physically. The next sections describe the conduit's design in detail.

DESIGN OF THE CONDUIT: LINKAGE AND FUNCTIONS

The conduit is located in the system LINKPACK and is always resident irrespective of whether CHAT and CPS are. It is activated early in system fire-up by a scheduled job written by Blair [B3]. This job places the conduit in the EXCP-intercept linkage chain which allows it to look for all EXCP (actually, "SVC Ø") issuances of interest to CHAT. Other components in the system, e.g., RJE and HASP, also intercept before the OS/360 SVC Ø logic gets a chance--hence, the linkage chain. Blair's program stores the address of the next interceptor (after the conduit) into the conduit's storage. The conduit uses this address for branching when an EXCP is not of interest. When intercepting, the conduit is in supervisor, interrupt-disabled mode.

Figure 7.1 shows a simplified overview of the conduit and the program elements interacting with it. The conduit has a CPS-side and a CHAT-side, which are packaged together and share common control storage. The CPS-side intercepts CPS EXCPs and analyzes channel programs directed to a CHAT "port." It stores information about the channel program in the control storage it shares with the CHAT-side and, after posting MIOS, branches back to CPS via the standard OS/360 code to end an SVC routine. CPS resumes execution at the instruction following EXCP. Part of the EXCP protocol is to pass the I/O Block (IOB) to the conduit. This control block contains the address of the channel program which defines the type of operation (e.g., read/write) and the location and length of data areas. The CPS-side posts the occurrence of the CPS-initiated event to MIOS via an event control block (ECB) in the station control block (SCB) associated with the port to which CPS directed the channel program. The

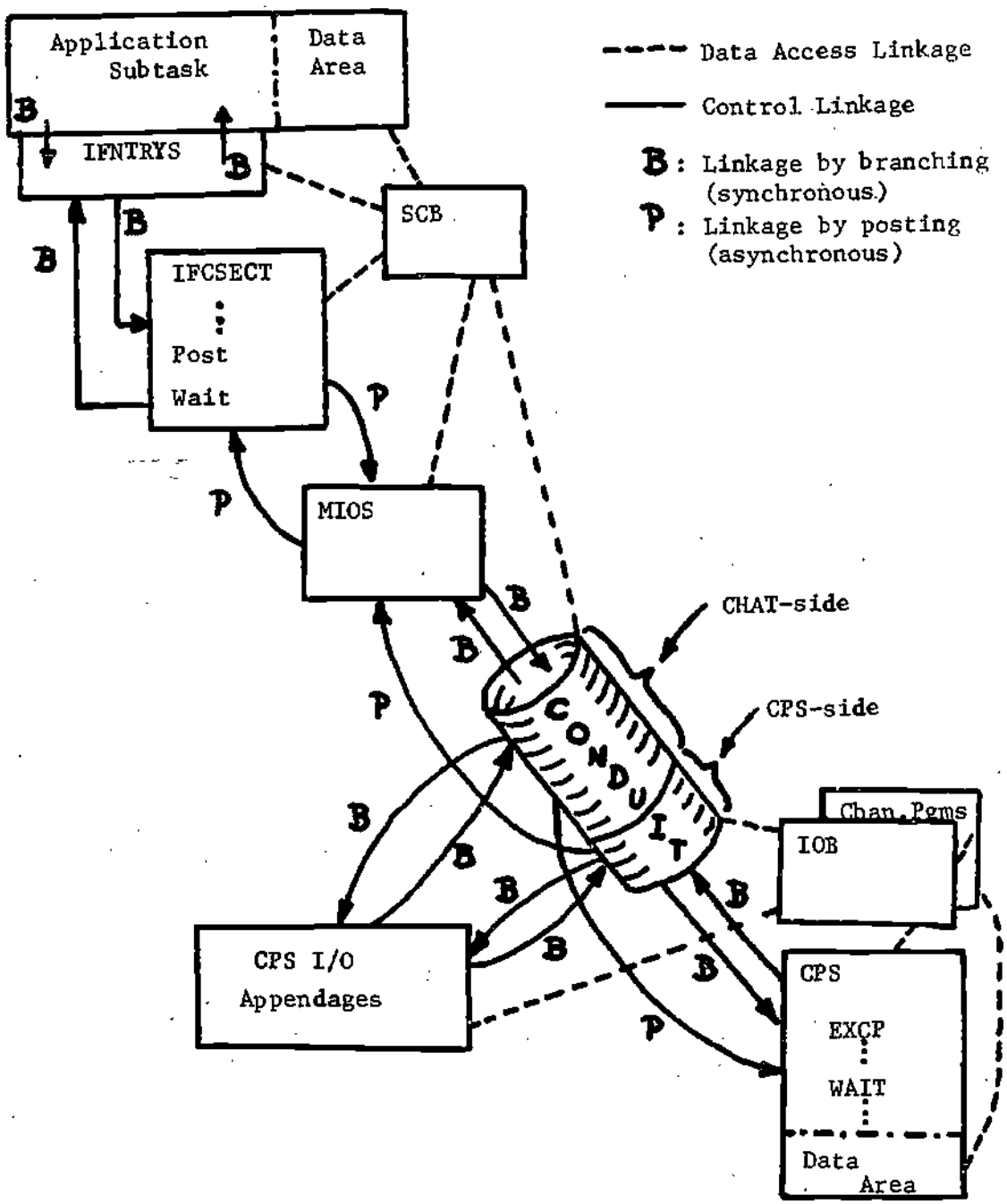


Figure 7.1 The Interregional Conduit

conduit gets addressability to the CHAT SCBs by another initialization action described later.

The use of the word port here stems from the presence in the system of actual Unit Control Blocks (UCBs) representing the fictitious physical channel addresses, or ports, defined for communication between CPS and CHAT. UCBs are the most primitive I/O control blocks in OS/360 and must exist before any I/O related activity can be done. Even though no physical I/O ever occurs using these UCBs, they are needed so that OS/360 will accept the DD cards in the CPS JCL and CPS's use of OS/360 services like OPEN and CLOSE. Because it has legal UCBs, CPS can use these services with gay abandon--we again did not have to change anything in its code. (Aside: Although CPS makes heavy use of Halt I/O, as will be explained, it only issues the System/360 HIO for a port once in-line, as part of initialization. Since System/360 allows HIO for a nonexistent physical address, without program-checking, even this did not require change.)

Currently, ten UCBs are defined as CHAT ports--more than the current number of display stations, to allow for growth. These contain the hexadecimal physical port addresses X'FØ' to X'F9'--the F being the way the CPS-side detects an EXCP to CHAT. Hence UCBs are useful to the conduit to detect and identify CHAT-port requests; they are found via the IOB (I/O Block). The UCBs are included in the system by a system generation process described by Blair [B3]. To save storage in CPS, the number of DD cards for CHAT ports is equal to the current number of display stations defined in CHAT, so some CHAT UCBs are not yet referenced by CPS.

Because of the posting by the CPS-side, the OS/360 task scheduler eventually activates MIOS, which has been waiting for any of the

sundry events it handles to occur. Synchronization of a CPS action with a CHAT application subtask action is a key concern. MIOS, therefore, first checks in the posted SCB to see if a field there indicates that the attached subtask has a request pending to communicate with CPS. If so, MIOS executes a branching protocol for direct entry into the CHAT-side of the conduit. Assume, however, that no request is pending: then MIOS ignores the post from the CPS-side and waits for a request from the application subtask. (Nothing can be done until the application program presents a compatible request, giving appropriate data pointers, etc.)

An application program may use any of the CPS-access procedures discussed in Chapter 3 and listed again in Table 7.1. A call invokes a corresponding subroutine in IFCSECT which puts the application program's data area address and length as well as the request type in the SCB, posts MIOS, and waits for MIOS to respond. RDCPS also issues an OS/360 STIMER macro specifying time, prior to waiting, and waits for either the message arrival or the time-out event to occur. The first event to occur causes RDCPS to cancel the other and to report the first one to the application via the return argument. If, for example, the time interval elapses, RDCPS sets the CPS-access request type in the SCB to zero, erasing the request. Thus if the CPS-side then posts MIOS, MIOS ignores it until the next application request.

When MIOS is posted by the IFCSECT routine, it checks whether the application subtask had previously been informed by ABNORM-signalling that CPS is dead. If so, MIOS does not allow the new request and posts the IFCSECT routine with a code requesting it to stop the violating application. If MIOS does not detect a violation it executes the branch-

Standard Log-On:	LOGCPS
Log-on with file name:	LOGCPS (libname)
Read from CPS	RDCPS (inarea, time, return)
Write to CPS:	WRCPS (text)
Attention, CPS:	ATNCPS

Table 7.1 CPS-Access Procedures

ing protocol to the CHAT-side of the conduit.

The branching protocol involves a change of program-state: MIOS executes an SVC (Blair's SVC 239 [B3]) giving it key-zero supervisory privilege. It then disables interrupts to create the same machine state for the CHAT-side that the CPS-side is given. This also allows the later-used CPS I/O Appendages to have the state which their coding assumed.

The CHAT-side has a choice of three return points in MIOS when it has completed handling the request:

1. Normal Return - exchange between the CHAT subtask and CPS was completed successfully.
2. Abnormal Return - One of the following errors occurred:
   a. WRITE INTERLOCK
   b. LOG-IN EXCEPTION
   c. CPS DEAD

   (See Chapter 3) A field in the SCB indicates which one.
3. CPS-not-ready Return - The CHAT request was not completed because CPS has not yet issued its next EXCP.

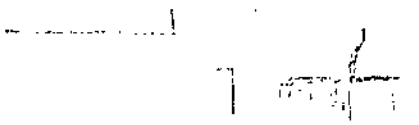
For each return, MIOS reverts to its normal interrupt-enabled, problem program state. For the normal return (1.), MIOS posts the IFCSECT routine and erases the request type in the SCB. The IFCSECT routine can then link back to its caller in normal fashion.

For abnormal return (2.), MIOS erases the request, turns on a flag if CPS DEAD is indicated (to remember that the application has been notified), and posts the special code to the IFCSECT routine requesting it to signal ABNORM to the application subtask. The code stored by the CHAT-side is picked up by ERRCODE when this procedure is called from the

application's on-unit (see Chapter 3).

For return (3.), MIOS does not erase the request and does not post the IFCSECT routine. Instead it waits for the CPS-side to post the occurrence of the next CPS EXCP for this port so it can retry the exchange.

This completes the discussion of the interactions of the CHAT components outside the conduit both with the conduit and among themselves. In the next section more is given about internal conduit logic and the internal relationships of the CPS-side and CHAT-side with each other and with the CPS I/O Appendages.



INSIDE THE CONDUIT

This section examines the inner workings of the conduit in greater detail: its initialization, its interactions with CPS and CHAT, and the cooperative activity between the CPS-side and the CHAT-side. Figure 7.2 shows the internal layout of the conduit and will be referred to in the following subsections.

Initialization and Checking: The discussion based on Figure 7.1 took for granted that the linkage mechanism was already in place: that CHAT could locate the conduit and that the conduit could locate both CHAT and CPS. In actual operation, the mechanism requires linkage-initialization as well as checking overhead during execution of the linkage. Initialization involves, primarily, making CPS known to the conduit, and CHAT and the conduit known to one another. Checking is performed to protect against improper linkage when CHAT is present but CPS is not, or vice versa--either because one has never been loaded into the system or because one has left the system before the other.

The two sides of the conduit experience no difficulty in communicating with each other: they are assembled in the same load module and can share commonly addressable control storage. In fact, one subroutine, for analyzing CPS channel programs, is used by both.

The CPS-side linkage with CPS is also easily established. Blair's program [B3], mentioned earlier, has a root in the conduit load module. This allows it access to the conduit control field into which it places the address of the next interceptor in the EXCP-intercept chain. This root is invoked at system fire-up, issues Blair's catch-all SVC 239 [B3], stores

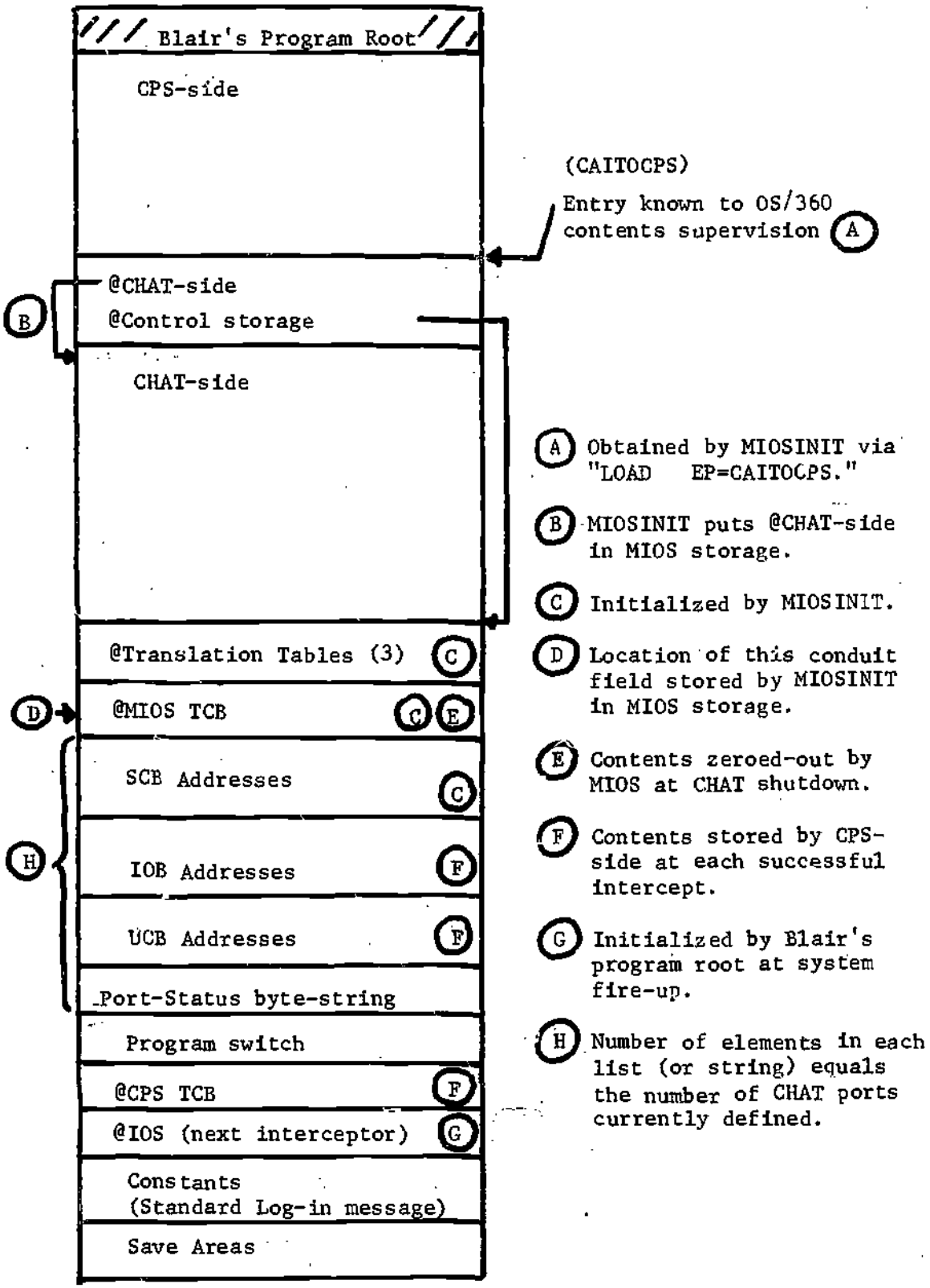


Figure 7.2 Internal Layout of the Conduit

the information returned (see Figure 7.2), and exits with its mission accomplished. CPS remains completely unaware of the interception mechanism and requires no knowledge of the conduit.

During subsequent operation, when the CPS-side successfully intercepts a CPS channel program to a CHAT port, the CPS-side obtains, as part of the standard EXCP protocol, the address of the OS/360 Task Control Block (TCB) for CPS and the address of the IOB that CPS has defined for the CHAT port currently in use.

The TCB address is useful for three reasons:

1. The CPS-side uses it at interception-time to determine quickly whether CPS issued the EXCP. It determines this by checking the program name associated with (and accessed through) the TCB address passed by the standard EXCP (SVC Ø) protocol. At TUCC, the CPS program has the tongue-twisting name: TUCSYSCPCPS. If it is CPS, with a channel program directed to a CHAT port, the CPS-side stores the TCB address in the conduit control storage. This is done each interception time, so that the conduit has the latest location (CPS may leave the system and come back again later).
2. Both conduit-sides need the CPS TCB address when posting CPS--the OS/360 posting protocol requires it.
3. The CHAT-side needs the CPS TCB address to check whether CPS is alive and well when CHAT (MIOS) requests access to CPS. However, because the CHAT-side and CPS operate asynchronously, the CHAT-side checking is somewhat involved. It may be that no CPS TCB address is stored in the conduit; in this case, CPS has not yet been intercepted by the CPS-side and cannot be located. Alternatively, a TCB

address may be stored but the TUCSYSCPCPS name-check fails; this indicates CPS has appeared in the system and then died (its TCB has been reused by OS/360). Finally, the name-check may succeed but certain bit-settings, or flags, in the TCB may indicate CPS is currently in the process of termination. Any of these negative indications prevents current use of CPS, causing the CHAT-side to return to MIOS with the CPS DEAD status.

The IOB address intercepted by the CPS-side establishes the conduit's means of accessing CPS. Figure 7.3 shows the detailed linkages involved; some of the linkages, such as that concerning the Start I/O Appendage and that for Halt I/O (HIO IOB), are described in a later subsection.

The CPS-side stores the IOB address into the conduit control storage (Figure 7.2) in the element of the IOB address-list indexed by the UCB hexadecimal X'Fx' identifier shown in Figure 7.3. The zone digit (F) in the hexadecimal-identifier informs the CPS-side that CPS is using a CHAT port. The CPS-side extracts the x-digit from the hexadecimal-identifier, multiples it by four, and uses the result to index the corresponding element of the conduit IOB list in which to save the current IOB address. Thus, X'FØ causes storing in the first fullword element of the list; X'F1', in the second; and so forth. (This design constrains the maximum number of CHAT ports to 16--more than we believed would ever be needed for the actual CHAT System. The design could be extended to allow another zone to be defined and checked, thereby increasing the number to 32--the maximum number of displays allowed by the CHAT display multiplexer.)

The same indexing applies to the conduit's list (Figure 7.2) of CHAT

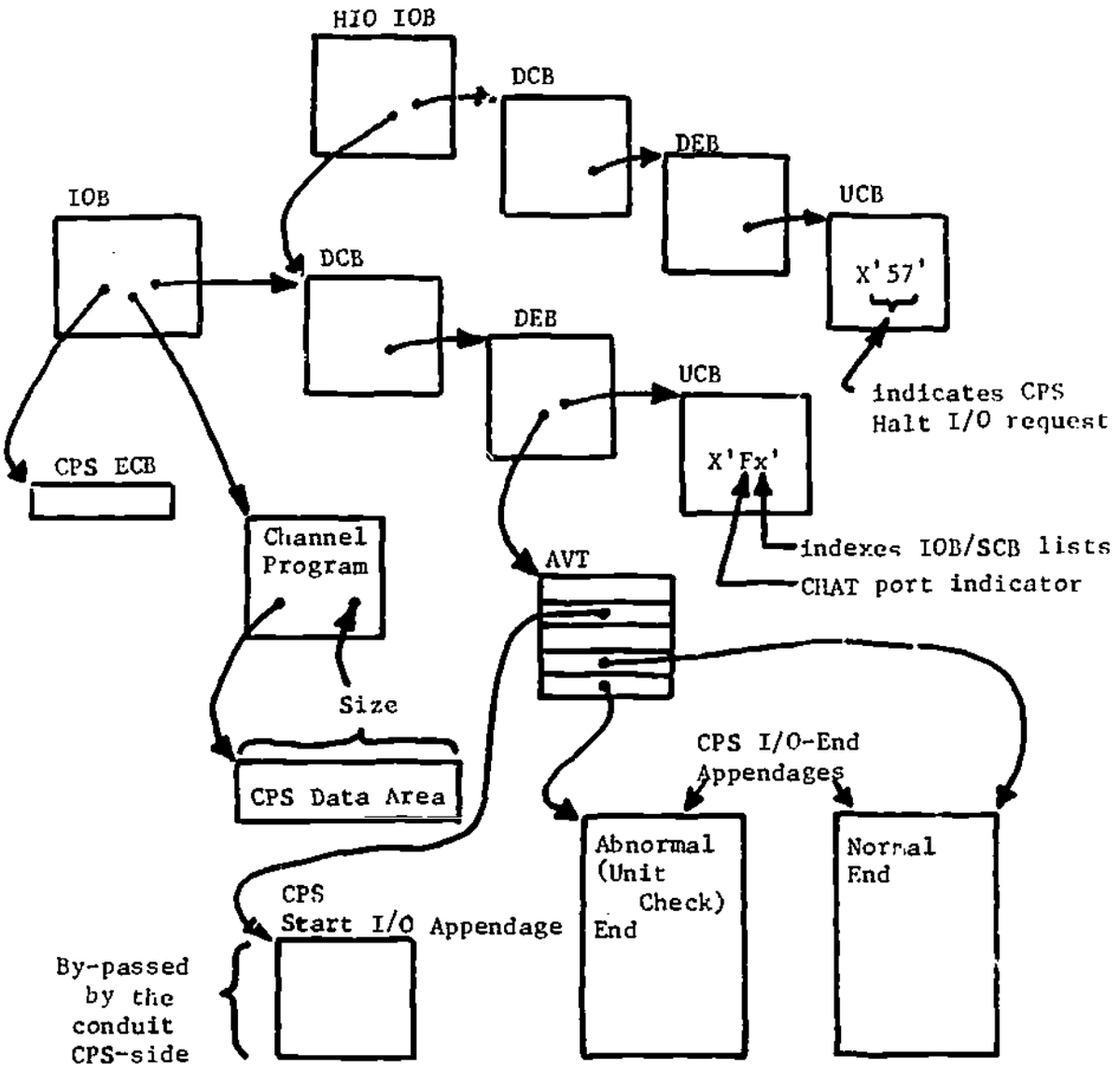


Figure 7.3 IOB Linkage to CPS

station control blocks (SCBs); the initialization of this list we have not yet described. Each SCB contains the same identifier, without the zone, for quick indexing by the CHAT-side when it needs to pick out the IOB address associated with the SCB address passed to it by MIOS. Figure 7.4 illustrates this correspondence, assuming for brevity only three ports. (The same principle applies to the conduit's list of UCB addresses; but because this list was useful primarily during debug testing of the conduit, we omit further mention of it in the thesis.) This association of SCB with IOB is the means the conduit uses to connect a specific application program to a specific CPS CHAT-port and to control concurrent activity at all ports. (Note: The index shown in Figure 7.4 is distinct from that defined in Chapter 5 in Figure 5.2.)

The IOB permits full access to CPS from the conduit. Notice (Figure 7.3) that it allows posting of CPS (the CPS ECB), access to CPS data areas (via the CPS channel programs), and access to CPS I/O appendages, which are discussed later. The Data Control Block (DCB), Data Extent Block (DEB), and Appendage Vector Table (AVT) are OS/360-defined control blocks. They are not of interest here apart from their intermediate role in the linkage and will not be described.

The SCB provides access in the other direction. The CHAT-side receives an address of an SCB through the MIOS branching-protocol described in the previous section. Fields in the SCB describe the type of request and the location and size of application program areas. An MIOS ECB (one that MIOS waits to have posted) is defined within each SCB. Hence, the CPS-side can post an ECB in a particular SCB to simultaneously wake up MIOS and signify to MIOS which CHAT port is active. MIOS can then, if needed, enter the CHAT-side, passing the address of the active SCB.

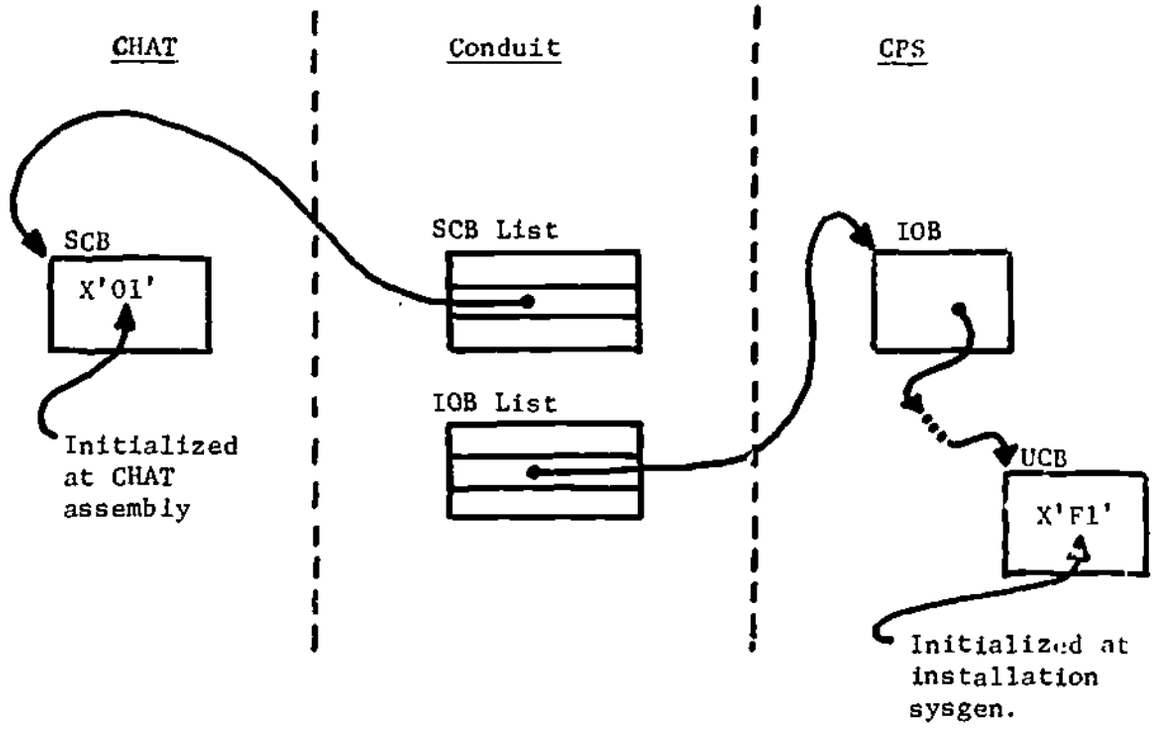


Figure 7.4 SCB and IOB Linkage

Making CHAT and the conduit known to each other is accomplished during the initialization of the CHAT region by an initialization routine named MIOSINIT. MIOSINIT is attached by the MSS control task and thus represents (temporarily) one of the three monitor control tasks. When it finishes its work, MIOSINIT transfers control (XCTLs) to MIOS which inherits the TCB used for MIOSINIT: MIOS becomes the resident control task in place of MIOSINIT and MIOSINIT, by using XCTL, disappears from the CHAT region. (It is transient.) While active, MIOSINIT performs various initialization functions: among them our present concern.

MIOSINIT locates the conduit using the services of OS/360 contents supervision, as indicated in Figure 7.2. A name, CAITOCPS, of a parameter list is declared an ENTRY and its address is returned when MIOSINIT specifies it in an OS/360 LOAD macro. This is all the addressability MIOSINIT needs in order to do the initialization. Into the conduit control storage, it stores its own TCB address (the one inherited by MIOS) and the list of SCB-addresses correctly ordered according to index. The addresses of three translation tables are also placed in the conduit. These tables are resident in the CHAT Monitor control storage but are used by the conduit for EBCDIC-to-Teletype and Teletype-to-EBCDIC code translation and for Teletype control character stripping. The two code translation tables are shared by the CHAT Monitor control task, MTWX. (MTWX uses a different table, however, for control character stripping and parity checking.)

MIOSINIT also initializes the CHAT Monitor control storage used by MIOS. There, it stores the address of the entry point to the CHAT-side of the conduit so MIOS can branch to it. MIOSINIT also stores into MIOS control storage the address of the conduit field in which it stored its

own TCB address. This allows MIOS as part of its shutdown protocol to zero-out this field in the conduit so that the CPS-side will know CHAT is not present. (MSS does this as part of its shutdown protocol if MIOS has abnormally terminated.) All of this initialization activity by MIOSINIT (and that by MIOS or MSS in zeroing-out the MIOS TCB address) is performed in supervisor, interrupt-disabled mode.

The CPS-side, which is asynchronous with respect to CHAT, has simpler checking to perform because of this shutdown cleanup. Before posting MIOS, it checks only whether the MIOS TCB address is present in the conduit. If it is, then the CPS-side can post MIOS (passing the TCB address to OS/360) using the, now addressable, SCB of interest. If the TCB address is not present, it will not post but instead will force the currently active CHAT port to the disconnected state--a process to be described. Similarly, the other active CHAT ports (if any) are forced to disconnect as their activity is intercepted. If CHAT is later restored, the MIOS TCB address will again appear in the conduit and be seen by the CPS-side.

This completes the detailed discussion of the checking and linkage-initialization required for the conduit to provide robust exchanges between CHAT and CPS. It also illustrates how elaborate mechanisms are sometimes needed to perform a function not provided for (indeed, "prohibited") within the host control program.

How CPS Works: Since the conduit uses the Teletype I/O interface presented by CPS, we need to describe briefly how CPS normally uses it, in communication with a real Teletype. Some familiarity with the CPS external specifications manual [I1] on operator usage of a CPS Teletype might be

helpful to the reader here, but not at all crucial. The following description is based on the author's line-by-line tracing through CPS source code listings--no other reference can be cited.

CPS uses seven distinct channel programs for the Teletype. These will be characterized by their five main functions:

Enable - This channel program is the one used by CPS to answer a dialed call from a Teletype. It is the first one issued to a port and does not complete (it "hangs") until a call is received. The channel program includes a write command that causes the familiar CPS "hello" type message to be printed at the calling Teletype.

Disable - This simply disconnects the Teletype. CPS follows this with an Enable channel program to await a new call.

Read - This is used by CPS to read keyed-in data from the Teletype.

Write - This is used by CPS to print out data on the Teletype. CPS uses three versions of this channel program, depending upon how it wants to control Teletype carrier-return/line-feed. Each version involves a different channel program structure.

Prepare - This channel program is used by CPS to monitor the line from the Teletype when no other channel program is available (ready). This permits CPS to learn that the operator has hit Break or the X-On key (attention) when no other channel program is monitoring line activity. Completion of a Prepare causes only I/O status to be signalled; no data is received (read).

In using the OS/360 EXCP interfsce, CPS also defines three I/O appendages: two for handling channel-end/device-end (CE/DE) I/O interrupts and one Start I/O appendage (see Figure 7.3). A Start I/O appendage is given control by the OS/360 I/O Supervisor (IOS) after an EXCP (SVC Ø) has been issued but before the actual System/360 Start I/O instruction is executed by IOS. Since a Start I/O appendage, like the I/O interrupt-handling appendages, has supervisor, interrupt-disabled privileges, it can be used to do some things not allowed the problem-state program issuing the EXCP. Notice that the mechanism allows for intercepting all EXCPs for which the appendage is defined [I7].

Start I/O appendages can have various uses--CPS uses its Start I/O appendage to stop I/O! This rather exotic usage (not copied by the CHAT Monitor support for the Teletype which borrows other features of the CPS Teletype support) stems from the programming complication introduced by the added function that the Prepare channel program provides. To maintain continuous surveillance over the connected Teletype, CPS constantly alternates EXCPs for the Prepare channel program with those for Read and Write channel programs that do work scheduled by the CPS user program or by the CPS Interpreter itself. Each of the latter CPS programs experiences processing delays, during which the Prepare channel program is needed to be active on the System/360 channel. However, this means that an incomplete Prepare must be removed in order to do the subsequent, scheduled I/O.

CPS employs a somewhat complicated technique which we simplify here. On completion of a Read or Write operation, CPS issues a new EXCP, passing the standard IOB shown in Figure 7.3 and using the Prepare channel program. Then, when scheduled Read or Write work is to be performed,

CPS fills in a bogus IOB--shown in Figure 7.3 as the HIO IOB. Both the IOB used for the Prepare and the HIO IOB are examined by the CPS Start I/O Appendage when it gains control. The former is passed through to IOS for Start I/O execution. The latter is trapped by the Start I/O appendage. Through linkage not shown in Figure 7.3 (which is oriented toward the conduit's concerns), the Start I/O appendage manages to locate the Prepare IOB-to-UCB chain and issues a System/360 Halt I/O, setting a flag (called IOBHALT) in the IOB. The Start I/O appendage then returns to the problem program, by-passing the IOS Start I/O execution.

When the interrupt caused by the Halt I/O occurs, one of the CPS I/O appendages is invoked by IOS interrupt-handling. On detecting that the IOBHALT flag is ON, the I/O appendage, through some other devious linkage not shown in Figure 7.3, locates the newly scheduled Read or Write channel program, fills in the (real) IOB, and uses the exit to IOS for restarting I/O--in this case the new channel program. The Halt I/O caused interrupt is not posted to CPS. Later, when the Read or Write successfully completes, the I/O appendage takes the exit to IOS specifying to post CPS. The cycle then continues.

Figure 7.3 shows linkage involving the HIO IOB. This is defined within CPS, but its use is more germane to the conduit. CPS includes a DD control card in its job-initiation JCL that specifies a real, sys-genned UCB with the unit identifier X'57'. This is to satisfy OS/360 validity-checking (the bogus IOB must be linked to a legal UCB), since CPS never links to it. The UCB itself represents an address of a fictitious physical device, just like CHAT ports.

Simulating the Teletype: The CPS operation, while exotic, is a felicitous design for the purposes of the conduit: it allows the conduit to simulate the operation of the Teletype without change to CPS code.

The implementation of Halt I/O through a Start I/O appendage allows the conduit to intercept the process, if CHAT ports are involved, before the appendage is reached. The presence of a specially marked (X'57') UCB, unique to this purpose, allows the intercepting CPS-side to detect the requirement. One of the CPS "clever" linkages--the pointer in the HIO IOB to the true DCB (shown in Figure 7.3)--allows the CPS-side to find the real UCB affected. This is the only CPS-unique linkage used by the conduit. (CPS puts the pointer to the true DCB in the IOBSTART field of the HIO IOB--a field normally used for a pointer to the channel program to be started.) When the CPS-side finds that the true UCB identifies a CHAT port, it uses the x-digit in the UCB to index the conduit IOB list where the true IOB address is already stored--since the previous Prepare channel program had been intercepted in the regular way.

In intercepting CPS EXCPs for CHAT ports, the CPS-side completely by-passes the CPS Start I/O appendage, although CPS activity for non-CHAT ports continues to invoke it. Both sides of the conduit do make use of the CPS I/O appendages, however, and in fact share usage of the same copy that serves concurrent CPS activity unrelated to CHAT.

Figure 7.5 gives a simplified view of how the CPS-side responds to a CPS EXCP to a CHAT port. If the CPS EXCP specifies the true IOB, the CPS-side obtains the address of the channel program from the IOB and examines the channel program to determine which one of the seven it is. A Disable is handled by the CPS-side without assistance from the CHAT-side. The CPS-side enters the CPS I/O appendage, with I/O inter-

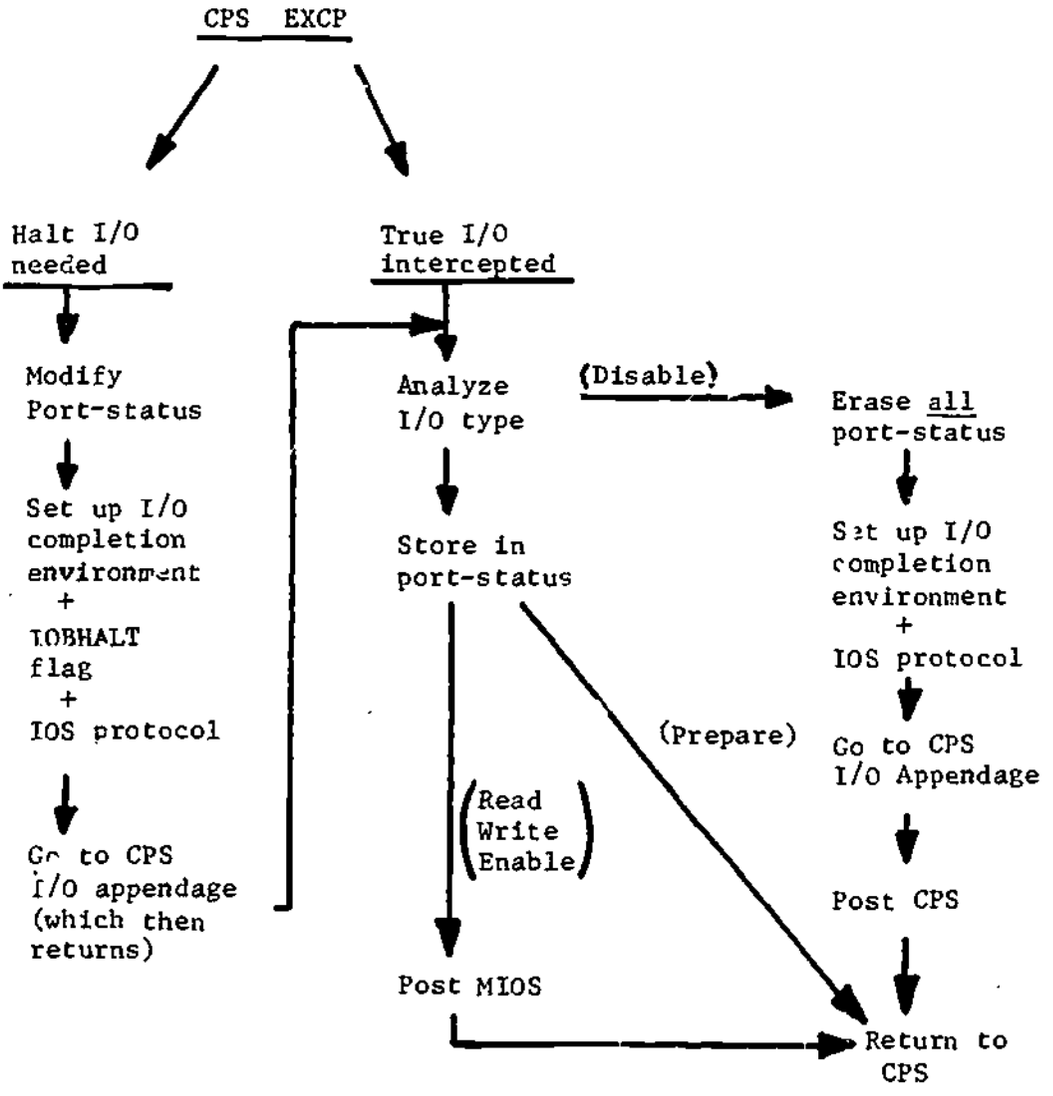


Figure 7.5 Simplified View of CPS-side Logic

rupt status simulated; the appendage processes the I/O completion interrupt just as it does for a real Teletype and returns to the CPS-side using its standard IOS-defined exit protocol (here, the exit signifying to post). The CPS-side then invokes the OS/360 posting protocol, which posts CPS, and the CPS-side then returns to CPS at the location following its EXCP-issuance. (Amusingly, this means the I/O completion has been posted before CPS is aware that its EXCP has finished. Since I/O completions are normally asynchronous with the program's sequencing, this causes no difficulty--even though real ones cannot occur in interrupt-disabled mode.)

The other channel programs are handled differently. A Prepare normally causes the CPS-side simply to update the conduit's records for the port (IOB address, current status) and to return to CPS. Read, Write, and Enable are of interest to CHAT, so the CPS-side posts these events to MIOS (if present) and then returns to CPS. These channel programs, unlike Disable, remain pending until a CPS Halt I/O request (to stop the Prepare) or a CHAT-side event occurs.

A Halt I/O request is handled in more involved fashion. The CPS-side creates the proper environment for a halted channel program interrupt and invokes the CPS I/O completion appendage to process it. The appendage uses its standard clever way of locating the new channel program and returns to the CPS-side via the IOS-defined restart exit, with the IOB properly re-initialized. Since this must be in accordance with IOS-defined protocol, the CPS-side easily locates the new channel program and goes to the analysis logic as shown in Figure 7.5.

Simulating the Teletype involves also simulating the System/360 channel (and the outboard control unit) and IOS, insofar as CPS append-

ages are sensitive to their operation. The invoked CPS appendage must find System/360 channel status word (CSW) and IOS-defined information of the standard format and in the standard locations in the IOB (IOS places the CSW information in the IOB, as one of its standard services). Similarly, the conduit must adhere to the standard IOS protocols on entry to and exit from the appendages, since these are the ones assumed in the appendage-coding. This involves initializing standard registers with standard pointers on entry and providing the exit support defined by IOS. The appendages must also find flags set the way the by-passed CPS Start I/O appendage would have set them if it had been invoked. (Besides IOBHALT, there is also an IOBBUSY flag not described here.)

The port-status referred to in Figure 7.5 is shown in greater detail in Figure 7.6. Each CHAT port is represented by a byte in the conduit's port-status byte-string shown in Figure 7.2. Here again the familiar indexing tactic applies: each CHAT port is represented by the byte in the string indexed by the x-digit in the SCB and UCB associated with it. Thus, the status for the port associated with the SCB/IOB pair shown in Figure 7.4 is kept in the second byte of the string.

The meaning and usage of the individual bit-fields of the status are as follows:

Bit Ø - This is turned ON by the CHAT-side to request that the CPS-side force the port to the disconnected state. A port is disconnected when CPS is awaiting a new dial-in. This means it has an Enable channel program pending.

Bit 1 - This is turned ON when the CHAT-side simulates completion of an Enable channel program. It remains ON until a Disable occurs. It is useful for informing the CHAT-side of the

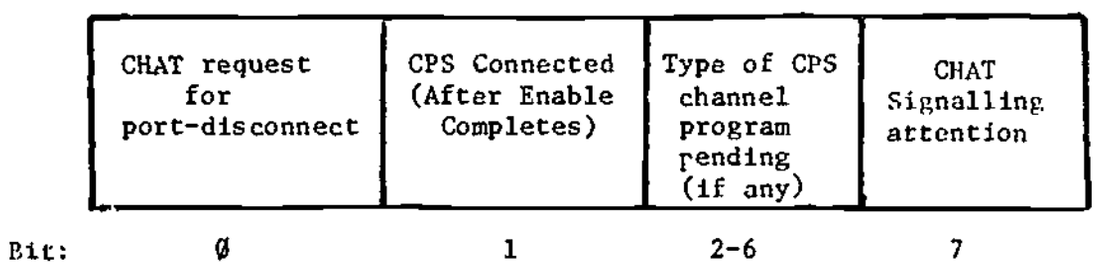


Figure 7.6 Port-status Byte

connection status even though no CPS channel program may be pending when the CHAT-side is invoked. (Note that CHAT and CPS compete for CPU-usage; so CHAT itself can be the cause of CPS's delay.)

Bits 2-6 - These bits inform the CHAT-side of the type of channel program currently pending (if any). They are set by the channel program analysis routine and reset when an I/O completion is simulated.

Bit 7 - This is set ON by the CHAT-side when the CHAT application subtask has issued ATNCPS at an instant when no channel program is pending. It alters the subsequent intercept-activity shown in Figure 7.5 where the Prepare handling is concerned: the CPS-side will execute the branch to post MIOS.

As indicated in Figure 7.5, the CPS-side sets all bits to zero when it intercepts a Disable.

The CPS-side intercept-activity is also altered from the way shown in Figure 7.5 if Bit Ø is ON: the handling of Read, Write, and Prepare is changed. For these channel programs, the CPS-side executes I/O completion simulation and does not post MIOS. (MIOS is not interested.)

For Read, this involves creating the environment associated with arrival of the Teletype EOT control character and System/360 unit exception status in the CSW. The normal-end CPS appendage (see Figure 7.3) is invoked and, upon return, the CPS-side posts CPS and returns to CPS.

For Write and Prepare, the CPS-side creates the completion environment associated with line-breakage (unit check status, intervention-required sense-bit, and IOS error-indicators) and invokes the abnormal-

end CPS appendage. The CPS-side then posts CPS and returns to it. Unfortunately, the machine-status itself is ambiguous, being also signalled for the case where a Teletype operator hits the Break key when these channel programs are active. Thus, CPS will assume the non-fatal cause and invoke its own Break-protocol. This involves a new Write for which the CPS-side on the new EXCP interception must persistently signal line-breakage, since the CPS abnormal-end appendage, at this time, will repeatedly return by the IOS-defined restart exit until its retry count is exhausted. Only then does CPS accept the fatal cause and give up. (The process can be further complicated because CPS extends "attention" handling to CPS user programs in its language definition, even permitting nested attention on-units. Thus, a number of Write EXCPs can be issued until the innermost attention-handler (CPS) is reached. Only on this innermost one does CPS retry to exhaustion.)

The CHAT-side also participates in the Teletype simulation process and does all the work to move data between CPS and CHAT. Figure 7.7 shows a simplified view of how the CHAT-side works. MIOS enters the CHAT-side for an additional reason beside those previously mentioned--to force the port to be disconnected if it is not already so. MIOS does this whenever an application subtask associated with the port terminates or if it is performing shutdown.

The conventions of Figure 7.7 are:

- The all-capitalized names correspond to CHAT requests to the conduit--either from the application subtask (LOG, READ, WRITE, ATN) or from MIOS for the disconnect reason (DISC). The request type is obtained by the CHAT-side from the SCB.
- The exits refer to those previously described for returning to MIOS.

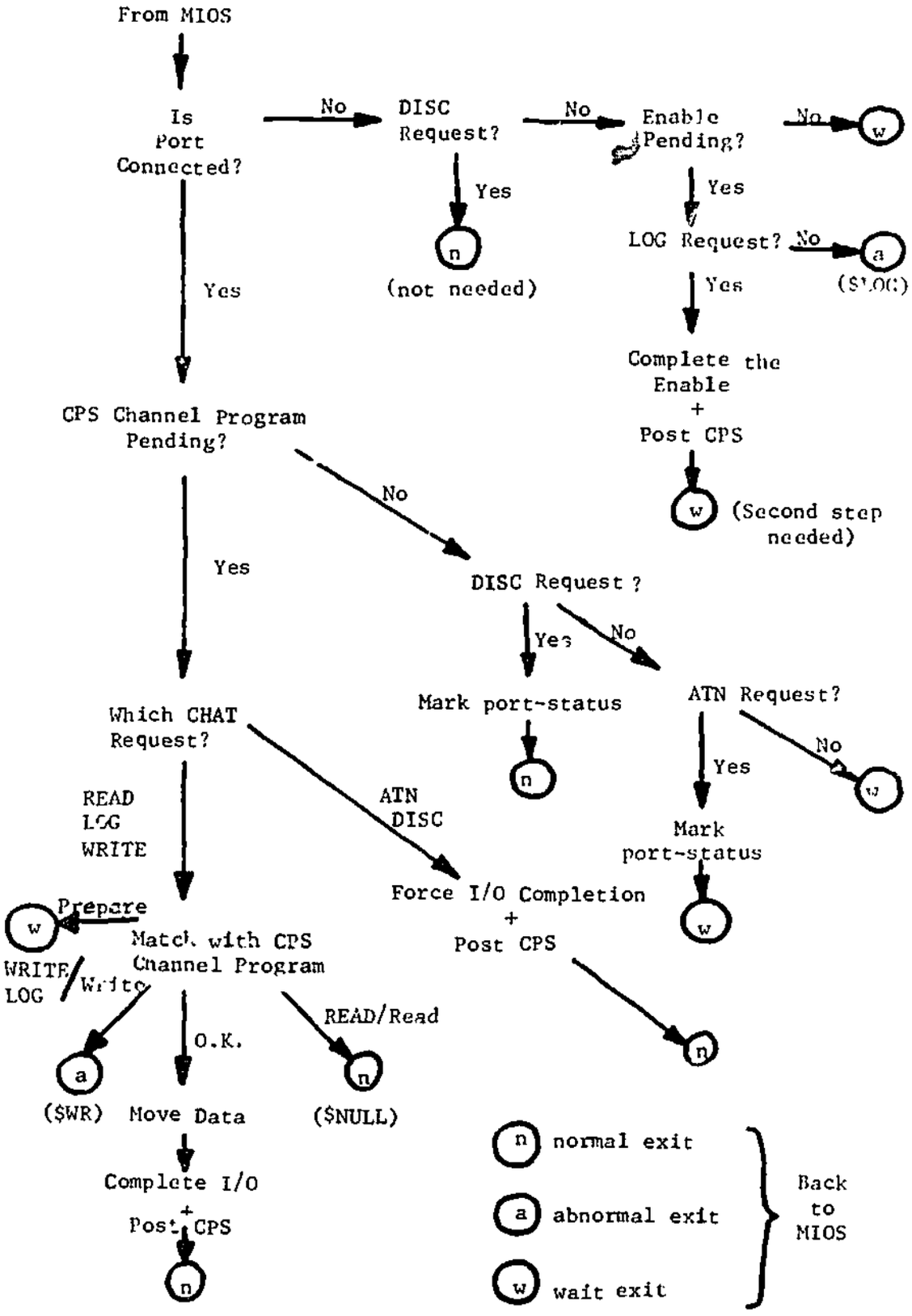


Figure 7.7 Simplified View of CHAT-side Logic

- Reference to the CPS channel programs is as before--capitalized first letter.
- The $-symbols specify the type of abnormal condition detected:

  $WR - both CHAT and CPS are trying to write to each other (CHAT loses).

  $LOG - the application program needs to log-in to CPS because CPS is awaiting a new call (not connected).

  $NULL - both CHAT and CPS are trying to read from each other. This could happen if a CHAT application subtask is reading from CPS and letting this indicator inform it of the end of a sequence.

- The comments along the arrows are obvious, except possibly those from the match-test: these indicate which CPS channel program is pending and the nature of the mismatch.

Notice that the CHAT-side completes the simulation of the Teletype I/O for those channel programs that the CPS-side left pending. The CHAT-side completes I/O by also using the CPS appendages, in the same way described for the CPS-side. However, because it executes the movement of data between CHAT and CPS, it also has Teletype code and control characters to manipulate. Code translation is required in both directions, while Teletype control character deletion/addition depends on direction. The conduit has no intermediate buffer, but moves data directly from the CHAT application program data area specified in the SCB to the CPS data area specified in the channel program--or vice versa. All code translation, of course, is performed at the sink after moving. Truncation (on the right) is performed if the source-length exceeds the sink-length.

Certain steps shown in Figure 7.7 require additional comment. The

LOG request involves two invocations of the CHAT-side. When it is issued by the application program, the port will normally not yet be connected: an Enable channel program is pending. Since CPS does no reading in this channel program, the CHAT-side must complete it first (the "hello" message is not moved into CHAT) and wait for the CPS-side to post MIOS that the next channel program has been intercepted. This one will be a CPS Read, and the CHAT-side can then move the application program's log-in message (the standard conduit-supplied log-in message with or without an application program specified load/save file) into CPS's data area. The application program does its own log-off using the interface facilities for writing to CPS. If the program writes a log-off/resume message [I1], it can issue a LOG that will take only one invocation of the CHAT-side to complete. (It can also use a WRITE to format its own log-in message.)

Notice in Figure 7.7 that the CHAT-side attempts to initiate disconnection of the port if a channel program is pending when the DISC request arrives. It marks the request for the CPS-side even in this case (not shown in Figure 7.7), so that if any further attempts are necessary the CPS-side will continue them.

The ATN request causes the CHAT-side to indicate the request in the port-status if no channel program is pending. This is to force the CPS-side to post MIOS on a Prepare so that the application program can get CPS's attention even in this case. The indicator is turned off by the CPS-side as it posts. When a channel program is pending, the CHAT-side does not indicate the ATN request in the port-status but honors the request itself. This means on a Prepare or Write that the condition described earlier as ambiguous is set for the simulated I/O completion.

This time the subsequent CPS Break-protocol Write can succeed. For Read, the simulated completion is made to appear as if the X-On (Control-Q) key has been hit on a Teletype [11]. This is unambiguous.

EXPERIENCE WITH THE CONDUIT

The original purpose of the conduit was to serve Mudge's DIAL program, as described at the beginning of this chapter. This purpose, however, was never realized for two reasons: (1) the move of the CHAT System from TUCC to UNC resulted in both a critical reduction in available core size and no other community needs for CPS in the new CHAT host installation; (2) Mudge's program was found not really to require usage of CPS. The UNC version of the CHAT System does not use the conduit.

The conduit was, however, completely tested and fully debugged at TUCC. In addition, Blair [B3] wrote an application program using the PL/I conduit interface which proved the conduit's soundness and usefulness. This program was invokable from any display station and provided display station access to CPS, since Blair used both the conduit CPS-access interface facilities as well as those for the display in his program. This ingenious application program has also been lost to the CHAT System because of the move. The CPS program at TUCC has reverted to its former staid existence of serving only real teletypewriter terminals.

CHAPTER 8: ON-LINE TERMINAL TEST FACILITY

Early experience with the display equipment gave pungent evidence of the need to include a diagnostic and testing facility within CHAT. During the development of the CHAT Monitor, the display equipment frequently broke down. Restoring it to working order was inconvenient, expensive, and, during one lengthy period, nearly hopeless. The problem was worsened at this period (at TUCC) by the communication lines and AT&T 201B1 data sets (involving three different telephone companies) used to connect the CC-7012 to the CC-72. The common-carrier equipment did not offer Swiss-watch reliability: within a year, three line breakages occurred and five data set replacements were necessary.

Bell gives responsive service--in part, because its management randomly "bugs" and tapes customer calls to Bell service centers. Bell diagnosis of data sets consists of running a series of set-to-set tests using special equipment with myriad lights to check that particular, repeated data patterns are correctly received at each end. Maintenance consists of replacing a defective data set by a spare that the service man carries along; if the Bell tests show no error, the data set is not replaced and the problem is blamed on the other vendor(s).

Unfortunately, the tests are not as exhaustive as they appear: After "successful" completion of Bell's testing of a data set, the author noticed a subtly uneven cursor-motion on the display screen and suggested to the service man that the data set crystal oscillator timer was bad

(the CCI equipment derives its transmission timing from the data set). The service man confessed that he had no test for this component in his standard on-site repertoire; some ad hoc strobing by the service man proved the oscillator was awry--the data set was replaced.

Diagnosis and servicing of the CCI equipment was hindered by the following deficiencies:

1. No hardware service aids other than the oscilloscope and strobes. This meant an almost total reliance on software control in the computer and on the standard (possibly defective) installed equipment to diagnose problems.
2. Ignorance on the part of the servicing personnel of the logical properties of the CCI equipment registers and of the communication protocols involved in data exchange (bit-x in the CC-72 SS-register means "this" after "that" action). Hence, strobing only established voltage levels and did not easily pinpoint logical problems (such as transmission parity block-checks).
3. Mis-design of the CC-72 multiplexer (no line-testing mode)--it requires computer program control in order to send anything, even the signal that one of its stations is ready to send a message. The computer program, of course, requires perfect action by the CC-7012 and communication equipment for its enabling order to arrive at the CC-72. (Scenario: Author-to-Bell: I think something is wrong with our equipment and I'm not sure whether it's you or CCI causing the problem. I thought I'd call you first. Bell-to-author (later): Well go ahead and send something; we're monitoring the line. Author-to-Bell (brief pause): Er--uh, well, I can't; my local equipment requires computer control before it can send and,

well, I have no on-line computer support (see next point). Can't you send your man out with that box with all those lights?...)

4. Poor manufacturer-supplied software diagnostic support. This support was grossly inadequate for use by the manufacturer service personnel and not at all useful to the CHAT designers who could never get documentation on what it did or what its output meant. Furthermore, this diagnostic support offered no interaction with the CPU console to allow operator control or inquiry, but communicated only with one of the unreachable displays. Its output, if any, was printed on the installation printer in cryptic format. The most crippling aspect of the support was that it was a standalone (card-loaded) program coded to run on the basic System/360 machine. Hence, it could only be used when the installation management was willing to shut down the operating system.

One scarring episode illustrates the gravity of the above deficiencies. During development of the CHAT Monitor when no code had yet been successfully tested for the displays, the author noticed that the equipment didn't work and reported the problem. Six months later, after innumerable weekends and mid-night hours of fumbling about and not isolating the error, the author decided the outage was too persistent. Even the author's installation-acceptance test program was dusted off during this period--being better at times than nothing! The equipment was disinstalled from the System/360 and shipped back to the California factory. There they found errors in both the CC-72 and the CC-7012; meanwhile, Bell--acting on a request for re-testing of its equipment--located a problem in the UNC-based data set.

This pattern of inadequate diagnosis and service was an important

lesson during the design period: the CHAT System had to provide a testing facility in order to be a viable production system. CHAT does include such a facility, centered in the on-line testing program, OLTEST (On-Line Test)--a PL/I program invokable from the Teletype. This chapter is about OLTEST--its objectives and usage, its special needs with regard to CHAT Monitor extensions, its interactions with the CHAT Monitor, its I/O programming interface, and the test command set and output presented at the Teletype.

Blair gave major assistance in the actual coding of OLTEST. He devised the basic scanning technique in OLTEST for parsing commands and also suggested inclusion of exercising-commands beyond those intended by the author. The facility is more ambitious and the human factors quality higher as a consequence of his interest and participation.

OBJECTIVES AND USAGE

Two general requirements had to be met by the testing facility for the CHAT System. It had to provide:

1. A diagnostic feature for use at the time an error or outage became visible to a display operator. What kind of errors were seen by the CHAT Monitor? What kind of I/O was involved? By knowing both the external symptoms and internal machine status of an error one might quickly diagnose the cause or specific location of the problem.
2. An equipment-exercising feature for use at the time of maintenance or service. In step with a current service diagnostic tactic, it should be possible to initiate a particular operation, to exercise a specific component, or to send a chosen character string.

The Teletype was the obvious console from which to control these features and so OLTEST was designed as a Teletype application program running under the control of the CHAT Monitor. This capability for control and requesting of test results at the site of the displays is very convenient. Depending upon the CHAT region-size at the time, it is possible to run OLTEST concurrently with other application subtasks. Thus, if only one display station is down, it can be tested on-line while the other displays are in production use.

OLTEST provides a set of commands for testing a display chosen by the Teletype operator and for requesting a status or log report. These commands and the output logging are described later in this chapter. The use of OLTEST for exercising the equipment should be easy for anyone who can use the Teletype. Interpretation of the log output for diagnostic

purposes is easier the more the observer understands about the operation of the display equipment and the channel programs used by MIOS. In actual practice, one visiting CCI service man, who examined the log output from OLTEST and compared it to the MIOS channel program it referenced, understood easily what error was occurring. (CCI service personnel understand channel programming.)

THE I/O INTERFACE FOR OLTEST

OLTEST had one special requirement that the application program interfaces described in Chapter 3 do not allow: access to both the Teletype and the display equipment from one program. Furthermore, OLTEST needed to choose a display from the group as currently directed by the Teletype operator. Both requirements, as well as an additional need to access Monitor storage for status-logging, are met by a special-purpose interface.

OLTEST includes in its source the preprocessor statement:

```
%INCLUDE (OLTDCL);
```

where OLTDCL is a member name in SYSLIB. This causes inclusion of the declarations needed to define the $-named variables and procedure entries for all the procedures described in Chapter 3 (except those for CPS access, which OLTEST doesn't need) as well as three special procedures described below.

OLTEST in using this interface is provided the special assembler-coded linkage routine, OLNTRYS, mentioned in Chapter 2. This differs from the linkage routines provided other application programs by including more than linkage to IFCSECT, where the display and Teletype procedures reside. The procedures to execute the new functions required for OLTEST are packaged with OLNTRYS (in the OLTEST load module) so they are resident in the CHAT region only when OLTEST is. Recall that IFCSECT is packaged with the Monitor and is always resident.

In describing the new procedures the same format is used as in Chapter 3.

CALL SEIZE (station, return);

The CHAR(2) station argument specifies the numeric identifier of the display for which all future display procedures invoked by the program are to apply. The identifier is the same as that appearing on the door of the room in which the display resides. (This should correspond to the "J-connection" numeric on the back panel of the CC-72 multiplexer where the station is attached.) The return argument (a BIT(8) ALIGNED variable) is set by the called procedure as follows:

'0000 0000'B - The specified station has been seized successfully.

'0000 0001'B - The specified station has not been seized because it is busy--some other application subtask is currently in use there.

'0000 0010'B - The specified station has not been seized--no display in the CHAT System has the specified station identifier.

After successful seizure, the program is in seized-mode. To escape from seized-mode the program invokes RELEASE:

```
CALL RELEASE;
```

This no-argument procedure reference causes release of the currently seized station (if any) and causes the program to escape seized-mode. It is a No-Op if the invoking program is not in seized-mode.

Two rules are enforced by OLNTRYS:

1. The program must be in seized-mode at the time it invokes a procedure associated with display station activity.
2. The program cannot invoke SEIZE when already in seized-mode. (It should release the previous one first.)

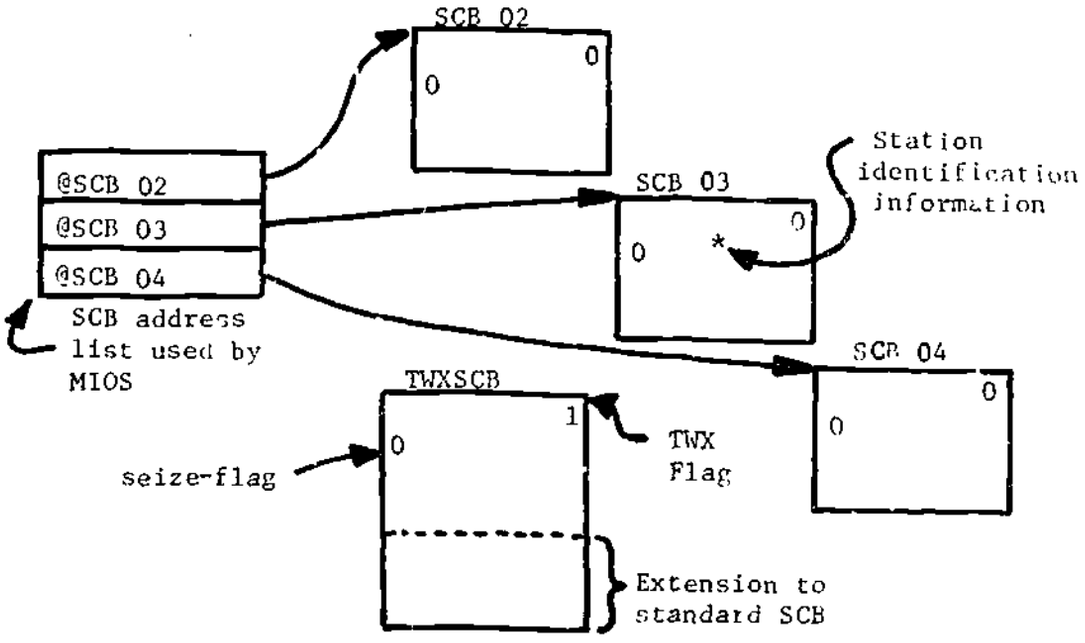
OLNTRYS enforces these rules by stopping the violating program at the time of the offending call.

The internal implementation of seizure of a display is simple: OLNTRYS, after finding that the specified station exists and is free, temporarily enters supervisor state and disables interrupts so that it can modify an address list used by MIOS. The process is illustrated in Figure 8.1 where we assume for brevity that there are only three displays. The address list is used by MIOS at times when it must locate a particular SCB indirectly (on I/O completions) rather than directly (on I/O requests where the posting task passes the SCB address). OLNTRYS also moves some station-specific information for addressing from the target station's SCB into the Teletype SCB and sets a flag indicating seized-mode exists. This flag is examined elsewhere in the Monitor: MTWX uses it to clean up after the application if it terminates without releasing the seized station; MIOS will not initiate termination action for the application using the SCB. (You cannot ABORT OLTEST from a display.)

Notice that the figure shows that the address of the true SCB for the seized station is stored in the Teletype SCB extension. This allows the release logic in OLNTRYS to un-do seizure. The extension also contains (not shown) the address of the Monitor's data-storage; this is how OLNTRYS finds the list it modifies. (The verification-of-station-id process also requires this list.)

The effect of this list-modification is very powerful considering its simplicity. OLTEST gains full access to the display using the standard display procedures and the same internal mechanisms. MIOS is insensitive to the SCB-switch in its handling of requests.

(a) Before Seizure



(b) After Seizing Station 03

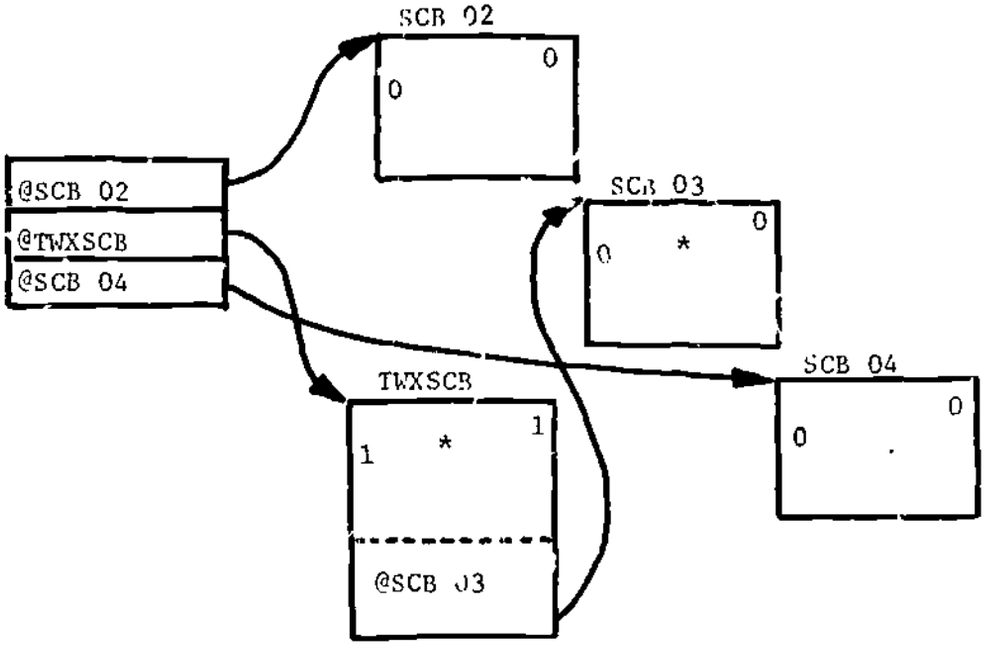


Figure 8.1 Seizure of a Display

The following procedure may be invoked regardless of mode. It provides the means whereby OLTEST obtains current I/O status information about equipment and malfunctions.

CALL ERRINFO (mios-flags, type-io, codes, sense, stat-id, return); If the BIT(8)ALIGNED return argument has been set by ERRINFO to the value '0000 0000'B, then a hard-failure record was found to have been stored by MIOS; all the other arguments will carry information from that record. The ERRINFO procedure (logic in OLNTRYS) will also have unlocked the record-storage used by MIOS to allow MIOS to store a new record. MIOS only stores information on one hard-failure at a time. When its record-storage has been unlocked it re-uses it for the next hard-failure to occur, locks it, and waits for the locked record to be retrieved before storing a record on any new hard-failure. Hence, ERRINFO retrieves the record of the first hard-failure to occur since the previous invocation of ERRINFO. This means the following sequence is useful:

(1.) Invoke ERRINFO to clear any old record.

(2.) Invoke a display procedure to perform some operation of interest.

(3.) Invoke ERRINFO afterwards to obtain the hard-failure record for the display operation if it failed. (This clears the record, too.)

(4.) Go to (2.).

In the later section on OLTEST commands the reader will learn how the Teletype operator can direct OLTEST through this sequence and get the results at the Teletype.

If no hard-failure record is found by ERRINFO (the record-storage

is not locked), it sets the return argument to '0000 0001'B and returns values in the mios-flags and type-io arguments (nothing in the other arguments) about on-going MIOS I/O activity at the instant ERRINFO was invoked. This information is useful in detecting "hanging" I/O, as will be discussed in more detail.

The arguments of ERRINFO have the following attributes:

mios-flags: BIT(8) ALIGNED

type-io: FIXED BIN(15,0)

codes: CHAR(*)

sense: CHAR(*)

stat-id: CHAR(2)

return: BIT(8) ALIGNED

The codes and sense arguments have their lengths governed by the value of a variable, MAXFAIL, which is part of the compile-time-included source. Currently, it is initialized to 7, corresponding to the MIOS-defined failure threshold (maximum number of failures). This is an assembled parameter (also called MAXFAIL in MIOS) that determines how many errors MIOS will tolerate on a single operation before giving up and treating it as "hard." Each failure in the sequence leading to the threshold causes MIOS to store information defining the failure. The codes and sense arguments correspond to two byte-strings kept by MIOS in its hard-failure record. The first byte in each string corresponds to the first failure; the second byte, to the second failure, and so forth. Hence, the length of each string is also governed by the maximum number of failures in a single sequence (MAXFAIL).

For return set to zero (hard-failure record returned), type-io will be set to one of the values shown in Table 8.1. These values specify

Value	Channel Program
0	GET STATION INTERRUPT STATUS (SIS)
4	GET LIGHT PEN CHARACTER
8	ENABLE KEYBOARD
12	GET LIGHT PEN COORDINATES
16	INPUT ALPHANUMERIC Part 1 if mios-flags = 'XOXXXXXX'B Part 2 if mios-flags = 'X1XXXXXX'B (X = not relevant)
20	OUTPUT ALPHANUMERIC
24	PERFORM SLIDE ACTION
28	(Reserved for Card Reader--Never Implemented)
32	READY-FOR-ATTENTION
36	VERIFY BREAK-FUNCTION
40	RESPONSE TO BOGUS BREAK (NO "ABORT" ON SCREEN)

Table 8.1 Values Possible for type-io

which channel programs were executed at the time of the hard-failure. These channel programs can be found in the code listing for MIOS; the names in the table correspond to comments heading each channel program listing. The mios-flags argument has relevance only for one particular channel program as shown also in Table 8.1. (That channel program has two parts--each requiring its own MIOS-initiated EXCP.) Possible values for each byte of codes are shown in Table 8.2. Although defined as a character-string, codes requires use of the PL/I feature, UNSPEC, since it is really a hexadecimal string. As shown in Table 8.2 the sense argument has relevance only in certain bytes in correspondence to "offset" bytes in codes. In this case, a byte in sense will be the sense-byte read in for the given failure. The stat-id argument gives the identifier of the display station involved during the hard-failure sequence or, if none (in the case of a multiplexer operation--see values 0 and 32 in Table 8.1), the characters NA for not applicable.

When return is set to one (no hard-failure record found), mios-flags tells whether MIOS had a channel program working on the System/360 channel at the time ERRINFO was invoked (and it picked up the MIOS status byte):

mios-flags = '0XXXXXXX'B means inactive

= '1XXXXXXX'B means active

(X = Not relevant)

The type-io argument gives the most recently initiated channel program and can be any one of the values shown in Table 8.1. However, two other values are possible if the initialization of the CHAT region is still incomplete. This will happen when the initialization routine MIOSINIT has experienced a problem initializing the CC-7012 and CC-72. In this case one of the following values is given:

Value	Meaning
<X'10'	A zero-origin index of the position of a CCW within a channel program. The CCW failed with unit-check status. The corresponding byte in sense gives the sense information accompanying the unit-check.
≥X'10'	A failure code signifying the type of failure in the display operation detected by MIOS. It concerns an inaction by the remote multiplexer or displays to MIOS orders. The corresponding byte in sense has no meaning. Specific failure codes are shown below.
X'10'	Cursor not restored (4)
X'20'	Keyboard not enabled (8)
X'30'	Cursor not restored (16)
X'40'	No acknowledgment to slide action (24)
X'50'	Cursor not restored (24)
X'60'	No acknowledgment to MIOS-sent message (20)
X'70'	Keyboard not enabled (20)
X'80'	No acknowledgment of Break-rejection message (40)

Table 8.2 Values possible for codes (per byte) shown in hexadecimal. The numbers in parentheses refer to the channel programs involved and correspond to the values in Table 8.1.

254 - MIOSINIT has reached the failure threshold in its initializing operation.

255 - MIOSINIT has not yet had its initializing channel program posted complete--a case of hanging I/O.

OLTEST provides a special message at the Teletype for each of these two conditions (as indeed it does for all others described).

In the former case (254) the Teletype operator should enter the MTWX command $E72. MTWX will post MIOSINIT which, in turn, will transfer control (XCTL) to MIOS where the more powerful hard-failure logging is performed and accessible through OLTEST. MIOSINIT will log on the CPU console on request but does not need to on the Teletype because MIOS has this capability.

OLTEST has one privilege that no other application program has: when the ABNORM condition has been signalled and ERRCODE returns the value signifying LINE ERROR (see Chapter 3), OLTEST may continue to use the display equipment procedures. Other programs are stopped if they attempt further equipment use. This added privilege is necessary since OLTEST has its greatest use when the display equipment is producing hard failures. The privilege is implemented by OLNTRYS as part of its linkage to display procedures; it is one instruction to turn off a flag.

COMMANDS

OLTEST provides a set of commands for the Teletype operator to exercise the display equipment and to request a log-out at the Teletype of the error and status information obtained from the ERRINFO procedure. Naturally, OLTEST reformats the information given by ERRINFO and provides more readable copy. Some examples are shown in the next section.

Figure 8.2 shows a typewriter version of an actual Teletype listing of an interaction with OLTEST. Notice that the operator uses the MTWX $XEQ command to invoke OLTEST, receives acknowledgment from MTWX, and begins receiving from OLTEST.

OLTEST always prompts the operator when it is ready for a new command. After reading and executing a command, OLTEST will print DONE if the command does not result in any other output on the Teletype. As shown, the HELP command does produce output, giving a summary of the facility and instructions for its use.

Figure 8.3 shows the consequence of using HELP ALL. A lengthy and detailed exposition of the OLTEST commands is printed on the Teletype. A typewriter version of this listing will serve the same purpose here. We add only a few remarks:

1. Notice that OLTEST does not require the operator to release a seized station prior to seizing a new one. It does this for him automatically.
2. The permitted short forms of the commands are not too useful--they are difficult to remember.
3. OLTEST accepts SIEZE to mean SEIZE (!).

4. OLTEST provides some error analysis on operator abuse of the commands which is not shown here. No abuse ever causes program stoppage.

5. The operator should feel free to interleave MTWX $-commands with the OLTEST commands, particularly $E72 or $ABORTnn (to get rid of some other application subtask "hanging" on a display to be seized). $ABORT is useful to stop OLTEST itself if the operator is tired of seeing a long output sequence; he can quickly re-initiate OLTEST using $XEQ.

6. It might be useful to have a short version of the OLTEST program--one without the HELP procedure--to save storage (if concurrent use of other applications is desired in a smaller region). This should not be difficult to do, but it has not yet been implemented.

```
MTWX HERE...ENTER A COMMAND
?$XEQ OLTEST
COMMAND EXECUTED

==> ON-LINE TEST:  DATE:  08/13/71  TIME:  23:43:29 <==
ENTER COMMAND OR "HELP"
?HELP

*-*-* INTRODUCTION TO ON-LINE TEST *-*-*

OLTEST IS INTENDED TO BE USED TO:
  1. EXERCISE THE CCI DISPLAY STATIONS--PARTICULARLY
     IN CONJUNCTION WITH MAINTENANCE SERVICE.
  2. LOG OUT CURRENT I/O STATUS OR "HARD" FAILURE
     SEQUENCES STORED BY THE MONITOR.
  3. AID IN THE DIAGNOSIS OF EQUIPMENT PROBLEMS.

*-*-* LIST OF OLTEST COMMANDS *-*-*

  LOG  SEIZE  RELEASE  END
  SEND SLIDE  REPEAT   LCAR  PAUSE  READ  READLP

*-*-* ASKING FOR MORE HELP *-*-*

IF YOU DESIRE FURTHER INFORMATION ON USE OF THE
ABOVE COMMANDS, TYPE IN "HELP" FOLLOWED BY THE
COMMAND NAME(S) IN WHICH YOU ARE INTERESTED.
  EXAMPLES:   "HELP LOG"
              "HELP SEIZE,RELEASE,SEND"
IF YOU TYPE "HELP SENSE", YOU WILL GET A DESCRIPTION
OF THE SENSE BYTE ABBREVIATIONS USED ON LOG OUTS.
IF YOU TYPE "HELP ALL", YOU WILL GET A DESCRIPTION
OF ALL COMMANDS (AS WELL AS "SENSE").

*-*-* ENDING OLTEST *-*-*

SIMPLY TYPE IN "END" AT ANY TIME.

*-*-*-*-*-*

NO GO AHEAD AND USE OLTEST.
ENTER COMMAND OR "HELP"
```

Figure 8.2 OLTEST Output for the HELP Command

```
?HELP ALL

*-*-* LOG COMMAND *-*-*

WHEN YOU TYPE IN "LOG", OLTEST WILL PRINT OUT ONE
OF THE FOLLOWING:
  1. THE INFORMATION STORED BY THE MONITOR FOR A
     "HARD" FAILURE--A SEQUENCE OF 7 CONSECUTIVE
     FAILURES ON A SINGLE I/O OPERATION WITHOUT
     RECOVERY. BY GETTING THIS ERROR LOG PRINTED
     OUT, YOU SIMULTANEOUSLY "CLEAR" THE MONITOR'S
     RECORD OF IT. (YOU CAN CLEAR THIS RECORD WITHOUT
     PRINTING BY TYPING "LOG CLEAR".) THE MONITOR
     RECORDS INFORMATION ONLY ON THE FIRST "HARD"
     FAILURE OCCURRING SINCE THE LAST "LOG". THUS,
     YOU WILL COMMONLY USE "LOG" IN ALTERNATION
     WITH THE OLTEST COMMANDS THAT EXERCISE
     THE EQUIPMENT.
  2. IF NO "HARD" FAILURE HAS BEEN RECORDED, YOU
     WILL BE GIVEN THE MOST RECENT I/O STATUS
     OF THE MONITOR. THIS IS USEFUL IN THE CASE
     WHEN AN I/O OPERATION IS "HANGING" (ON A READ
     CCW GENERALLY), BECAUSE THE LINE IS DOWN OR
     THE CC-72 MULTIPLEXER IS DOWN OR OFF.

*-*-* LOG OUTPUT *-*-*

A CURSORY KNOWLEDGE OF THE I/O OPERATIONS AND
CHANNEL PROGRAMS (IN THE MONITOR MODULE "MIOS") IS
HELPFUL IN INTERPRETING THE LOG OUTPUT--PARTICULARLY
FOR "HARD" FAILURES WHERE CHANNEL COMMAND WORD (CCW)
OFFSETS ARE CONCERNED. FOR "HARD" FAILURES YOU ARE
GIVEN THE STATION INVOLVED (OR NA-NOT APPLICABLE-FOR
A MULTIPLEXER OPERATION), THE TYPE I/O (CHANNEL
PROGRAM) THAT FAILED, AND THE SEQUENCE OF 7
FAILURES. SOME FAILURES INVOLVE SO-CALLED "SENSE"
INFORMATION FOR SPECIFIC CCWS. IN THIS CASE,
THE CCW IS IDENTIFIED BY ITS (0-ORIGIN) POSITION
IN THE CHANNEL PROGRAM, WHILE THE SENSE IS SHOWN
IN ABBREVIATED MANNER (SEE BELOW).
```

Figure 8.3 OLTEST Output for the HELP ALL Command

```
*-*-* SENSE BYTE INFORMATION *-*-*

WHEN THE SENSE BYTE INFORMATION IS LOGGED OUT,
THE FOLLOWING ABBREVIATIONS ARE USED:
  CR = COMMAND REJECT
  IR = INTERVENTION REQUIRED
  BO = BUS OUT
  3? = BIT-3 OF SENSE (NOT USED BY CCI WHEN CC-72 PRESENT)
  DC = DATA CHECK
  OR = OVERRUN
  TO = TIME OUT
  7? = BIT-7 OF SENSE (NOT USED ON "UNIT CHECK" BY CCI)
(DETAILS ON THESE ERROR INDICATORS CAN BE FOUND
IN THE CCI MANUAL ON THE CC-7012.)

*-*-*-*-*-*


*-*-* SEIZE COMMAND *-*-*

THIS COMMAND MUST BE USED BEFORE YOU CAN USE
THE COMMANDS:
  SEND  SLIDE  REPEAT  LCAR  PAUSE  READ  READLP
TO EXERCISE A PARTICULAR DISPLAY STATION.
YOU TYPE IN "SEIZE" FOLLOWED BY A 2-CHARACTER
NUMERIC IDENTIFYING THE DESIRED STATION.
  EXAMPLE:   "SEIZE 04"
THIS EXAMPLE CAUSES STATION 04 TO BE SEIZED
IF IT IS FREE.  (YOU CANNOT SEIZE A STATION
WHICH IS IN USE FOR ANOTHER APPLICATION.)
THE 2-CHARACTER NUMERIC WILL AGREE WITH
THE "J-CONNECTION" NUMERIC FOR THE STATION
(SEE BACK PANEL OF THE CC-72 MULTIPLEXER).
YOU MAY CANCEL THE EFFECT OF A PREVIOUS "SEIZE"
BY TYPING A NEW "SEIZE" OR BY USING "RELEASE".

*-*-*-*-*-*
```

Figure 8.3 (Continued)

```
*-*-* RELEASE COMMAND *-*-*

THIS COMMAND CAUSES A SEIZED STATION TO
BE RELEASED. A SHORT FORM, "RLS", MAY BE USED.
  EXAMPLES:   "RELEASE"
              "RLS"

*-*-*-*-*-*

*-*-* END COMMAND *-*-*

THIS COMMAND ENDS THE OLTEST PROGRAM.

*-*-*-*-*-*

*-*-* SEND COMMAND *-*-*

THIS COMMAND IS USED TO SEND A MESSAGE
TO THE CURRENTLY SEIZED DISPLAY STATION.
  EXAMPLE:   "SEND 'ABCDEF'"
THE CHARACTER STRING WITHIN THE SINGLE
QUOTES WILL BE DISPLAYED ON THE DISPLAY
STARTING AT THE CURRENT CURSOR LOCATION.

*-*-*-*-*-*
```

Figure 8.3 (Continued)

```
*-*-* REPEAT COMMAND *-*-*

THIS COMMAND CAUSES THE SCREEN OF A SEIZED
STATION TO BE CLEARED AND THEN A SPECIFIED
CHARACTER TO BE DISPLAYED IN ALL POSITIONS.
  EXAMPLES:   "REPEAT A"
              "REPEAT  "
              "REPEAT"
THE FIRST EXAMPLE SPECIFIES "A" AS THE
REPEATED CHARACTER; THE SECOND SPECIFIES
"BLANK"; AND THE THIRD, "*", THE DEFAULT
CHARACTER. ONCE YOU SPECIFY "REPEAT",
THE PROGRAM REPEATS THE OPERATION EACH
TIME YOU SEND "X-OFF". YOU MAY CHANGE
THE CHARACTER BY TYPING A NEW CHARACTER
BEFORE YOUR "X-OFF". YOU END THE REPEAT
OPERATION BY TYPING "STOP" (OR "END"--IF
YOU WANT ALSO TO END THE OLTEST PROGRAM).

*-*-*-*-*-*

*-*-* SLIDE COMMAND *-*-*

THIS COMMAND CAUSES A SLIDE ACTION
TO BE PERFORMED AT THE SEIZED DISPLAY
STATION.
  EXAMPLES:   "SLIDE ON"
              "SLIDE 40"
              "SLIDE OFF"
A SHORT FORM, "SLD", MAY BE USED.

*-*-*-*-*-*
```

Figure 8.3 (Continued)

```
*-*-* PAUSE COMMAND *-*-*

THIS COMMAND ALLOWS YOU TO SPECIFY A TIME
INTERVAL FOR THE OLTEST TO PAUSE, AWAITING
AN INTERRUPT FROM THE SEIZED STATION.
AFTER YOU ENTER "PAUSE", TYPE SOMETHING
AT THE DISPLAY AND HIT INTERRUPT. (YOU
MAY ALSO LIGHTPEN THE STARTING LOCATION
OF THE INFORMATION TO BE READ, IF YOU
LIKE.) OLTEST WILL REPORT THE RESULTS
OF ITS READ OPERATION ON THE TELETYPE.
  EXAMPLES:   "PAUSE 10"
              "PAUSE"
THE FIRST EXAMPLE CAUSES OLTEST TO WAIT
10 SECONDS FOR AN INTERRUPT FROM THE
SEIZED DISPLAY; THE SECOND EXAMPLE CAUSES
IT TO WAIT INDEFINITELY FOR THE INTERRUPT
(SO DOES "PAUSE 0").

*-*-*-*-*-*

*-*-* READ COMMAND *-*-*

THIS COMMAND CAUSES OLTEST TO READ
FROM THE SEIZED DISPLAY.  TYPICALLY,
YOU WILL ENTER "READ" HERE; GO TO THE
DISPLAY--ENTERING A MESSAGE THERE
(WITH AN INTERRUPT); AND THEN RETURN
TO THE TELETYPE TO SEE OLTEST'S REPORT
OF THE NUMBER OF CHARACTERS READ.

*-*-*-*-*-*
```

Figure 8.3 (Continued)

```
*-*-* READLP COMMAND *-*-*

THIS COMMAND CAUSES OLTEST TO READ
THE LOCATION COORDINATES AND CHARACTER
YOU LIGHTPEN AT THE SEIZED DISPLAY.
TYPICALLY, YOU WILL ENTER "READLP" HERE;
GO TO THE DISPLAY--LIGHTPENNING THERE
(AND HITTING INTERRUPT); AND RETURN TO THE
TELETYPE TO SEE OLTEST'S REPORT.
A SHORT FORM, "RLP", IS ALSO PERMITTED.

*-*-*-*-*-*


*-*-* LCAR COMMAND *-*-*

THIS COMMAND IS USED TO POSITION THE CURSOR
OF A SEIZED STATION TO A SPECIFIED ROW
AND COLUMN ON THE SCREEN.
  EXAMPLE:   "LCAR(10,25)"
(THE CURSOR WILL BE POSITIONED TO ROW 10,
 COLUMN 25.)
ALWAYS SPECIFY 2 DIGITS FOR EACH COORDINATE,
USING A LEADING ZERO IF NEEDED.
  EXAMPLE:   "LCAR(05,01)"

*-*-*-*-*-*

ENTER COMMAND OR "HELP"
?
```

Figure 8.3 (Continued)

OUTPUT FOR THE LOG COMMAND

This section shows and discusses some examples of actual output given by the LOG command. While in all cases the examples are faithful typewritten reproductions of real Teletype sessions, the equipment errors illustrated were contrived--e.g., by disconnecting the equipment component involved.

Figure 8.4 illustrates several ideas mentioned earlier in the chapter. The figure illustrates the case where the CC-7012 channel adapter is not operational at the time the CHAT region is initiated. In this case, the initialization routine MIOSINIT has not successfully finished its work and, thus, has not yet transferred control to MIOS.

As the figure indicates, the Teletype operator is able to dial into CHAT and to invoke OLTEST--since MTWX is active. The output for the first LOG command reveals to the operator the existence of the incomplete-initialization status. The operator then enters the $-command (to MTWX), $E72, asking that MIOS be invoked to enable the equipment. (OLTEST does not see this command.) The operator then enters the second LOG command of the session and, this time, gets a full report on the nature of the problem, since MIOS--with all its hard error record-keeping capability--has done its job. MIOSINIT is no longer in the system, having transferred control to MIOS at $E72 time. Notice how OLTEST gives LOG output in descriptive fashion--converting all those codes received, via its use of ERRINFO, into more readable language.

The error logout shows, via the station identifier (NA) and type of I/O, that a multiplexer operation has failed. Ready-for-attention is the first channel program tried by MIOS when it initially gains control.

```
MTWX HERE...ENTER A COMMAND
?$XEQ OLTEST
COMMAND EXECUTED

==> ON-LINE TEST:  DATE: 08/04/71  TIME: 19:23:10 <==
ENTER COMMAND OR "HELP"
?LOG
NO "HARD" I/O ERRORS RECORDED SINCE LAST LOG REQUEST
LAST I/O STARTED: (HARDERR IN MIOSINIT; AWAITING DIRECTION)
WHEN INSPECTED, I/O WAS NOT IN PROGRESS

?E72
COMMAND EXECUTED
?LOG
==> ERROR LOG ON 08/04/71 AT 19:24:10 <==
ERR SYMPTOM---  STATION NA  TYPE I/O =READY FOR ATTENTION
 1   CCW= 0, SENSE=IR
 2   SAME
 3   SAME
 4   SAME
 5   SAME
 6   SAME
 7   SAME
?LOG
NO "HARD" I/O ERRORS RECORDED SINCE LAST LOG REQUEST
LAST I/O STARTED: READY FOR ATTENTION
WHEN INSPECTED, I/O WAS NOT IN PROGRESS
?$E72
COMMAND EXECUTED
?LOG
==> ERROR LOG ON 08/04/71 AT 19:25:06 <==
ERR SYMPTOM---  STATION NA TYPE I/O =GET SIS
 1   CCW= 0,SENSE=IR
 2   SAME
 3   SAME
 4   SAME
 5   SAME
 6   SAME
 7   SAME
```

Figure 8.4 CC-7012 Outage at CHAT-Initiation Time

The first error recorded shows that the first (zero-origin indexing) Channel Control Word (CCW) failed with Intervention Required (IR) detected in the sense-byte. The first CCW in this particular channel program is a command to the CC-7012. Intervention Required ending status means the CC-7012 is not "powered-up" correctly. The same error has persisted through attempts 2-7 as indicated.

The output for the next LOG command illustrates how the previous LOG command had cleared the hard error record--only the current state of I/O activity is reported. The final LOG output shows continued persistence of the problem. Get-SIS (Station Interrupt Status) is the standard channel program executed by MIOS in response to a $E72 request--usually when the CC-72 requires enabling, it also has a station-interrupt pending.

Figure 8.5 illustrates some additional aspects of OLTEST. For this example, we deliberately turned off the power at a display station to simulate station outage. Notice the OLTEST syntax- and context-error messages to the operator for SEIZE 7 and SEND 'X', respectively.

The response of OLTEST to the SEND 'HELLO' request is a consequence of its (1) using the DISPLAY procedure (of Chapter 3) to send HELLO to the display station and (2) being informed by the Monitor via ABNORM-condition signalling (also Chapter 3) that the operation failed. The I/O ERROR report is sent by the invoked on-unit of OLTEST to alert the Teletype operator immediately.

The subsequent LOG command output shows that the channel program for DISPLAY (to Station 06) failed persistently on the eighth (zero-origin, again) CCW with Time-Out (TO) indicated in the sense-byte. This particular CCW is the first Read command in the channel program that

```
?SEIZE 06
DONE
?SEIZE 7
** TWO CHARACTER NUMERIC STATION ID REQUIRED - REENTER **
ENTER COMMAND OR "HELP"
?SEIZE 06
DONE
?SEND 'HELLO'
** I/O ERROR ON STATION/LINE
ENTER COMMAND OR "HELP"
?LOG
==> ERROR LOG ON 08/04/71 AT 19:26:23 <==
ERR SYMPTOM---  STATION 06  TYPE I/O =OUTPUT ALPHANUMERIC
 1   CCW= 7,SENSE=TO
 2   SAME
 3   SAME
 4   SAME
 5   SAME
 6   SAME
 7   SAME
?RELEASE
DONE
?SEND 'X'
** INVALID REQUEST -- NO CC-30 STATION HAS BEEN
   SEIZED BY USE OF THE "SEIZE" COMMAND. **
ENTER COMMAND OR "HELP"
?LOG
NO "HARD" I/O ERRORS RECORDED SINCE LAST LOG REQUEST
LAST I/O STARTED: READY FOR ATTENTION
WHEN INSPECTED, I/O WAS NOT IN PROGRESS
?$SHUTDOWN
```

Figure 8.5 Display Station Outage

requires a display station response. Several errors could cause the symptom; but the fact that an earlier Read in the same channel program--requiring a CC-72 response--succeeded, narrows the problem to the display-side of the multiplexer. Observation of the CRT-screen at the time of failure, to detect whether a Write--preceding the failing Read--also succeeded, will further narrow the problem.

As a final example, Figure 8.6 shows the normal state of the CHAT System. The first LOG output reveals the usual state of I/O: MIOS is awaiting some event to occur and the link to the displays is idle and ready for either-direction use. The second LOG output is a rare occurrence: although CHAT does a great deal of I/O, the fraction of time used is very small and in practice an I/O-in-progress message is difficult to capture.

The final output in the figure also concludes the chapter.

```
?LOG
NO "HARD" I/O ERRORS RECORDED SINCE LAST LOG REQUEST
LAST I/O STARTED: READY FOR ATTENTION
WHEN INSPECTED, I/O WAS NOT IN PROGRESS
?LOG
NO "HARD" I/O ERRORS RECORDED SINCE LAST LOG REQUEST
LAST I/O STARTED: OUTPUT ALPHANUMERIC
WHEN INSPECTED, I/O WAS IN PROGRESS
?END
** OLTEST ENDED **
```

Figure 8.6 Normal State of the CHAT System

CHAPTER 9: FACTS AND FIGURES

This chapter covers two important points about the CHAT Monitor: how to change it and how big it is. The first point is particularly important for growth requirements.

From the outset of the design of the Monitor, it was anticipated that the CHAT System would grow in number of display stations attached to the remote CC-72 multiplexer. Every effort was made to free the CHAT Monitor of design sensitivities to the number of displays currently in the cluster--as Chapter 5 described in some detail. The next section describes the simple-to-use facilities provided to re-parameterize the Monitor when there is a change made in the display configuration.

The Monitor can be modified in other ways not described here. For example, it may be desirable at some future time to modify the time-slicing parameters which are located in Monitor control storage. This type of change, however, is more the interest and concern of the system programmer inheriting CHAT responsibility; and thus such matters are left to the code listing. Other matters such as system generation and library maintenance are described by Blair in his companion thesis [B3] on CHAT.

CHAT PARAMETERS AND HOW TO CHANGE THEM

Table 9.1 lists a number of CHAT parameters that were designed to be changed easily without affecting the logic of the CHAT Monitor. The Monitor code uses these parameters as implied--without assuming a fixed value for any of them.

NUMCC3ØS is the most important of them since it defines the number of display stations to be supported. As shown, its current value is 6. To change it, simply replace the current equate defined in the macro definition, GENPARM. GENPARM is used widely in the CHAT Monitor (in almost every component) to generate the shown names as well as others in common use.

If the number of displays is to be increased, an SCB must be defined for each new display to be added. A macro by the same name is defined for this purpose. It should be coded as follows:

label SCB NXT,ID=nn,CPSUCB=Øx,ETX=etx

A label is required and the currently used labeling convention is recommended. The NXT operand should be used to generate any new SCB. The ID operand specifies the address identifier widely mentioned throughout this thesis; it corresponds to the identifier of the J-connection on the back of the CC-72 where the new display is attached. The CPSUCB operand specifies the CHAT-port to be used in the conduit as described in Chapter 7. The ETX operand specifies the hexadecimal code for the ETX control character in use at the display. This operand may be omitted and Ø3 will be assumed. (One old display used 13.) The CPSUCB operand may also be omitted in which case ØØ is assumed.

The current SCBs are also defined by macro and are located in the

Name	Current Value
NUMCC3ØS	6
MAXFAIL	7
CMAXLEN	20
RMAXLEN	90
MAXDATAL	80

Table 9.1 Parameters in GENPARM

ROOTCAI source deck which includes the COMN macro defining the Monitor control storage and the TWXSCB (which also uses a variant of this macro). Place the new SCBs in their logical order with respect to the current ones. The TWXSCB must be the last SCB in the deck. (Note: A version of the SCB macro exists for generating the SCB DSECT--simply code "SCB D.")

The other parameters in Table 9.1 probably need never be changed. MAXFAIL specifies the maximum I/O failure threshold while the remaining parameters determine the Teletype buffering constraints in MTWX (see Chapter 6). The currently defined values appear quite satisfactory. (Aside: Actually, the Teletype support has an error threshold different from MAXFAIL, which MIOS uses. In fact, two are parameterized! One is defined for use by the Teletype appendages; the other, by MTWX. This is for inner- and outer-error loop control and the author does not recommend changing them. They are MAXERR and MAXAPERR, assembled only in MTWX.)

STORAGE REQUIREMENTS

Table 9.2 lists the true core usage by the various components jointly referred to as CHAT Monitor support in this thesis (with the Acceptance Test "globule" thrown in for good measure). All except OLTEST are coded in assembler language.

Table 9.3 lists the CHAT region requirements for residence of the CHAT Monitor load modules. Discrepancies are due to OS/360's 2K-block allocation.

Initialization Code

MIDF	108
MSSINIT	502
MIOSINIT	1605
MTWXINIT	252

Resident Code

MSS	2072
MIOS[1]	6024
MTWX	2632
MTOC [B3]	1024
IFCSECT	2568
Conduit	2176

Monitor Control Storage

COMN	2746
SCBs	128.N
TWXSCB[2]	328

Resident with Subtask Modules

IFNTRYS	176
IFNTRYAS [B3]	312
IFNTWXS	112
OLNTRYS	696

Others

OLTEST (PL/I)	56,200
CHAT SVCs [B3]	See [B3]
Acceptance Test	5656 (Not in CHAT)

[1]Includes the display buffer
[2]Includes the Teletype buffers

Table 9.2 Storage Usage

MSS+MIOS+MTWX+IFCSECT+MTOC	16K
Control Storage (N ≤ 8)	4K
Subpool Ø	2K
Subpool 252	2K
Abend Dump (Disk only)	2K
Total	26K

Table 9.3 CHAT Monitor Load Module in the Region

REFERENCES

[A1] Anderson, R. H., and Farber, D. J. Extensions to the PL/I Language for Interactive Computer Graphics. The Rand Corporation, RM-6028-ARPA, Santa Monica (January 1970).

[B1] Balzer, R. M. Ports--A Method for Dynamic Interprogram Communication and Job Control. The Rand Corporation, R-605-ARPA, Santa Monica (August 1971).

[B2] Beyer, William F., III, Black, Sylvia S., Hamlin, Griffith A., Jr., Mailliard, Margaret A., and Wright, William V. Pikaplot--Laboratory Project Final Report for Computer Science 101. University of North Carolina at Chapel Hill (May 1969).

[B3] Blair, William H. Master's Thesis. University of North Carolina at Chapel Hill (In preparation).

[B4] Brownlee, Edward H., Jr. PAMELA: An Interactive Assembler System for the IBM/360 Computer. Master's Thesis. University of North Carolina at Chapel Hill, 1970.

[C1] Carmody, S., Gross, W., Nelson, T. H., Rice, D., and van Dam, A. A Hypertext Editing System for the System/360. Pertinent Concepts in Computer Graphics, Faiman, M. and Nievergelt, J. (Eds.). University of Illinois Press, Urbana, 1969, 291-330.

[C2] Computer Communications, Incorporated. CC-30 Communications Station Reference Manual. CCI, Inglewood, California, 1968.

[C3] Computer Communications, Incorporated. CC-30 Installation Procedures Guide. CCI, Inglewood, California, 1968.

[C4] Computer Communications, Incorporated. CC-7012/CC-72/CC-30 Communications System Programming Manual. CCI, Inglewood, California, 1968.

[C5] Computer Communications, Incorporated. CC-7012 Channel Adapter Reference Manual. CCI, Inglewood, California, 1968.

[C6] Computer Communications, Incorporated. CC-72 Multiplexer Reference Manual. CCI, Inglewood, California, 1968.

[C7] Conway, Melvin E. Design of a Separable Transition-Diagram Compiler. Communications of the Association for Computing Machinery 6,7 (July 1963), 396-408.

[D1] Denning, P. J. Third Generation Computer Systems. Computing Surveys 3,4 (December 1971), 175-216.

[D2] Dennis, J. B., and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. Communications of the Association for Computing Machinery 9, 3 (March 1966), 143-155.

[D3] Dijkstra, E. W. The Structure of the "THE" Multiprogramming System. Communications of the Association for Computing Machinery 11, 5 (May 1968), 341-346.

[F1] Freeman, D. N., and Pearson, R. R. Efficiency vs. Responsiveness in a Multiple-Services Computer Facility. Proceedings of the Association for Computing Machinery 23rd National Conference. Association for Computing Machinery, New York, 1968, 25-34B.

[G1] Grant, Charles A. Command Communication between Processes. Ph.D. Dissertation. University of California, Berkeley, 1971.

[G2] Gwynn, J. W. CRT Terminal Access from High-Level Languages. 1972 Society for Information Displays International Symposium Digest of Technical Papers, Volume 3, Society for Information Displays, 1972, 46-47.

[I1] IBM Corporation. Conversational Programming System (CPS) Terminal User's Manual. IBM Form GH20-0758.

[I2] IBM Corporation. IBM System/360 Operating System: Basic Telecommunications Access Method. IBM Form GC30-2004.

[I3] IBM Corporation. IBM System/360 Operating System: Concepts and Facilities. IBM Form C28-6535.

[I4] IBM Corporation. IBM System/360 Operating System: Supervisor and Data Management Macro Instructions. IBM Form GC28-6647.

[I5] IBM Corporation. IBM System/360 Operating System: Supervisor and Data Management Services. IBM Form GC28-6646.

[I6] IBM Corporation. IBM System/360 Operating System: System Control Blocks. IBM Form GC28-6628.

[I7] IBM Corporation. IBM System/360 Operating System: System Programmer's Guide. IBM Form GC28-6550.

[I8] IBM Corporation. Introduction to the Real-Time Monitor (RTM). IBM Form GH20-0824.

[I9] IBM Corporation. System/360 Principles of Operation. IBM Form A22-6821.

[I10] IBM Corporation. IBM System/360, PL/I Reference Manual. IBM Form GC28-8201.

[L1] Lynch, W. C. Operating System Performance. Communications of the Association for Computing Machinery 15, 7 (July 1972), 579-585.

[M1] Mudge, J. Craig. Human Factors in the Design of a Computer-assisted Instruction Syatem. Ph.D. Dissertation. University of North Carolina at Chapel Hill, 1973.

[M2] Mudge, J. Craig. On Writing Reentrant Programs in PL/I. SACM Newsletter--a Publication of the University of North Carolina Student Chapter of the Association of Computing Machinery, Chapel Hill (November 1971), 2-3.

[O1] Oliver, Alfred. A Measurement of the Effectiveness of an Interactive Display System in Teaching Numerical Analysis. Ph.D. Dissertation. University of North Carolina at Chapel Hill, 1969.

[S1] Scherr, A. L., and Larkin, D. C. Time-sharing for OS. AFIPS Conference Proceedings, 37, 1970 Fall Joint Computer Conference. American Federation of Information Processing Societies Press, Montvale, New Jersey, 1970, 113-117.

[S2] Sneeringer, James. More on Writing Reentrant Programs in PL/I. SACM Newsletter--a Publication of the University of North Carolina Student Chapter of the Association of Computing Machinery, Chapel Hill (December 1971), 5-7.

[T1] Triangle Universities Computation Center. CPS Terminal User's Manual. TUCC Memorandum No. LS-55. January 8, 1969.

[W1] Wait, Ty. Conversion of the Hypertext Editing System from the IBM 2250 Graphics Terminal to the CC-30 Alphanumeric Terminal. Master's Thesis. University of North Carolina at Chapel Hill, 1972.

[W2] Weiler, P. W., Kopp, R. S., and Dorman, R. G. A Real-Time Operating System for Manned Spaceflight. IEEE Transactions on Computers C-19, 5 (May 1970), 388-398.

[W3] Wilkes, M. V. Time Sharing Computer Systems. American Elsevier, New York, 1968.

[W4] Witt, B. I. The Functional Structure of OS/360, Part II, Job and Task Management. IBM System Journal 5,1 (1966), 12-29.

APPENDIX A

List of Acronyms

This appendix lists some of the more obscure acronyms that appear in the text.

BTAM Basic Telecommunications Access Method

CCI Computer Communications, Inc.

CPS Conversational Programming System

CVT Communication Vector Table

DCB Data Control Block

ECB Event Control Block

EXCP Execute Channel Program

HASP Houston Automatic Spooling Program

HIO Halt I/O

INT Interrupt

IOB Input/Output Block

JCL Job Control Language

MIOS Monitor I/O Scheduler for displays

MSS Monitor Subtask Scheduler

MTOC Monitor Table of Contents

MTTY Monitor Teletype control task

OLTEST On-Line Test

PSW Program Status Word

RTM Real-Time Monitor

RTOS	Real-Time Operating System
SAS	Station Acknowledgment Status
SCB	Station Control Block
SIS	Station Interrupt Status
SS	Short Status
STCBE	Subtask TCB Element
TCB	Task Control Block
TSO	Time-Sharing Option
TUCC	Triangle Universities Computation Center
UCB	Unit Control Block
XCTL	Transfer Control