United States Patent [19]

Lotito et al.

[11] **Patent Number: 4,625,081**

[45] **Date of Patent: Nov. 25, 1986**

[54] **AUTOMATED TELEPHONE VOICE SERVICE SYSTEM**

[76] Inventors: **Lawrence A. Lotito,** 6625 Springpark Ave., Los Angeles, Calif. 90056; **Teresa D. Huxford,** 1822 Pandora Ave., #3, Los Angeles, Calif. 90025; **Ann L. Donaldson,** 2321 W. 232nd St., Torrance, Calif. 90501

[21] Appl. No.: **445,651**

[22] Filed: **Nov. 30, 1982**

[51] **Int. Cl.**[4] **H04M 3/38;** H04M 3/50

[52] **U.S. Cl.** **379/88;** 379/196; 379/211

[58] **Field of Search** ..... 179/18 B, 18 D, 18 DA, 179/5 P, 6.02, 6.17, 6.18, 6.09, 6.11; 360/32, 12; 364/513.5, 513; 381/36, 51; 370/60, 61, 62

[56] **References Cited**

U.S. PATENT DOCUMENTS

Re. 30,903	4/1982	Vicari et al.	179/27 FH
1,922,879	8/1933	Burgener	179/27 FH
2,685,614	8/1954	Curtin	179/27 FH
2,863,950	12/1958	Dunning et al.	179/27 FH
2,892,038	6/1959	Gatzert	179/27 FH
2,985,721	5/1961	Gatzert	179/27 FH
2,998,489	8/1961	Riesz	179/6.02
3,141,931	7/1964	Zarouni	179/6.11
3,146,310	8/1964	Jeffries et al.	179/6.07
3,197,566	7/1965	Sanders et al.	179/18 BE
3,273,260	9/1966	Walker	434/307
3,296,371	1/1967	Fox	381/51
3,510,598	5/1970	Ballin et al.	179/18 BE
3,519,745	7/1970	Colman	179/5 P
3,728,486	4/1973	Kraus	179/2 R
3,733,440	5/1973	Sipes	179/18 B
3,920,908	11/1975	Kraus	179/2 CA
4,117,270	9/1978	Lesea	179/18 BE
4,200,772	4/1980	Vicari et al.	179/27 FH
4,210,783	7/1980	Vicari et al.	179/18 FC
4,256,928	3/1981	Lesea et al.	179/18 BE
4,272,810	6/1981	Gates et al.	364/900
4,302,632	11/1981	Vicari et al.	179/27 FH
4,320,256	3/1982	Freeman	179/6.04
4,371,752	2/1983	Matthews et al.	179/7.1 TP

OTHER PUBLICATIONS

"Store & Forward Voice Switching", International Resource Development, Inc., Report #145, pp. 45–56, Jan. 1980.

"A Design Model for a Real-Time Voice Storage System", Hattori et al., *IEEE Trans. on Communications,* vol. COM-30, No. 1, Jan. 1982, pp. 53–57.

Barish, Bernard T. and Slattery, Paul J., "BISCOM: Rx for Internal Communications", *Bell Laboratories Record,* vol. 42, No. 6, pp. 175–180 (Jun. 1974).

Watson, Jr., R. E. and S. B. Weinberg, "Telephone Answering Services," *Bell Laboratories Record,* vol. 43, No. 12, pp. 447–450 (Dec. 1965).

Liske, W., "Remote Controlled Switching of the Telephone Message Service of the Deutsche Bundespost," *TN-Nachrichten* vol. 70, pp. 13–16 (1970).

Probe Research, Inc., "ECS Telecommunications, Inc., " *Proceedings of Voice Processing Seminar,* Sep. 15, 1982.

Probe Research, Inc., "Voice Message Service," *Proceedings of Voice Processing Seminar,* Sep. 15, 1982.

Probe Research, Inc., "Logic Labs, Inc." *Proceedings of Voice Processing Seminar,* Sep. 15, 1982.

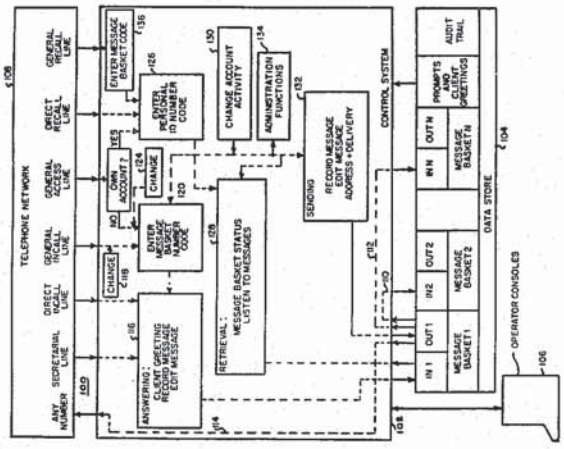
List Continued on next page.

Primary Examiner—Thomas W. Brown

[57] **ABSTRACT**

An automated telephone voice service system includes a data store having a plurality of addressable voice storage message baskets defined therein and a control system coupled between the store and a large plurality of telephone lines of a telephone network. An incoming cable may address a particular message basket by entering a code through the telephone keyboard or by a predetermined association with a particular call in line. Upon identification of the message basket the caller is greeted by a client's own voice and invited to leave a voice message which will be recorded in the message basket or given other client information. Upon entry of a personal identification code a caller is granted access to user account functions which include retrieval of voice messages, forwarding of messages to other message baskets or telephone lines, and administrative functions such as the changing of greetings or account operating criteria. Editing commands may be utilized during the recording of voice messages.

74 Claims, 27 Drawing Figures



Facebook Ex. 1016 Part 1
U.S. Pat. 8,243,723

OTHER PUBLICATIONS

Probe Research, Inc., "BBL Industries, Inc.," *Proceedings of Voice Processing Seminar,* Sep. 15, 1958.

Probe Research, Inc., "Wang Laboratories," *Proceedings of Voice Processing Seminar,* Sep. 16, 1982.

Probe Research, Inc., "American Telephone and Telegraph, Inc.," *Proceedings of Voice Processing Seminar,* Sep. 16, 1982.

Probe Research, Inc., "Commterm, Inc.," *Proceedings of Voice Processing Seminar,* Sep. 16, 1982.

Probe Research, Inc., "American Express Company," *Proceedings of Voice Processing Seminar,* Sep. 16, 1982.

Probe Research, Inc. "Equitable Life Assurance," *Proceedings of Voice Processing Seminar,* Sep. 16, 1982.

Probe Research, Inc., "Massachusetts General Hospital," *Proceedings of Voice Processing Seminar,* Sep. 16, 1982.

Seaman, John, "Electronic Mail Coming at You," *Computer Decisions,* pp. 129–160 (Oct. 1982).

"Voice Mail Update," *Electronic Mail & Message Systems,* vol. 4, No. 20 (Oct. 15, 1980).

Hanson, Bruce L., R. J. Nacon and D. P. Worrall, "Custom Calling Features Cater to Customers," *Telephony,* pp. 28–32 (Sep. 1980).

"Elect. Mail Pack Unveiled by DÉC." *Electronic News,* vol. 27, No. 1365 (Nov. 21, 1981).

ECS Telecommunications, Inc. Marketing Literature for their UMX System (Jan. 7, 1982).

Memo from C. W. Murphy to Jack Atkin Dated Jan. 30, 1981.

"ECS Unveils 1,000—User Digital Message Exchange," *Communications.*

Matthews, G. H., "The Pitfalls of Small Telecommunications Trunk Groups," *ECS Telecommunications, Inc,* (1981).

"New Product, Voice Message Systems," *Business Communications Review* pp. 37–40 (Jan.–Feb. 1981).

Dukes, A., "IBM Unveils Voice Mailbox; Seen as Step Toward PBX," *MIS Week,* vol. 2, No. 39 (Sep. 30, 1981).

"Speechfile—IBM's Secret Message System Weapon," *Electronic Mail & Message Systems,* vol. 5, No. 12 (Jun. 15, 1981).

"Introducing Voice Store & Forward," *Computer Decisions,* (Oct. 1981).

Out Voice Product Brochures, Voice and Data Systems, Inc.

Dukes, A., "Atlanta Firm Enters Voice-Message Arena," *Management Information Systems Week,* p. 6 (Nov. 18, 1981).

"New Local Net, Voice Store and Forward from Wang," *Computer Decisions* (Aug. 1981).

Delphi Delta 1 Telephone Operator's Training Manual (Apr. 1, 1981).

Delphi Delta 1 Voicebank Data Entry Reference Manual (Jul. 20, 1981).

Delphi Delta 1 Voicebank Marketing Literature.

Delphi Delta 1 Specification.

Delphi Delta 1 Standard Processor Module (SPM–1) Specification (Mar. 13, 1978).

Delphi Pascal Programmers Manual (May 22, 1981).

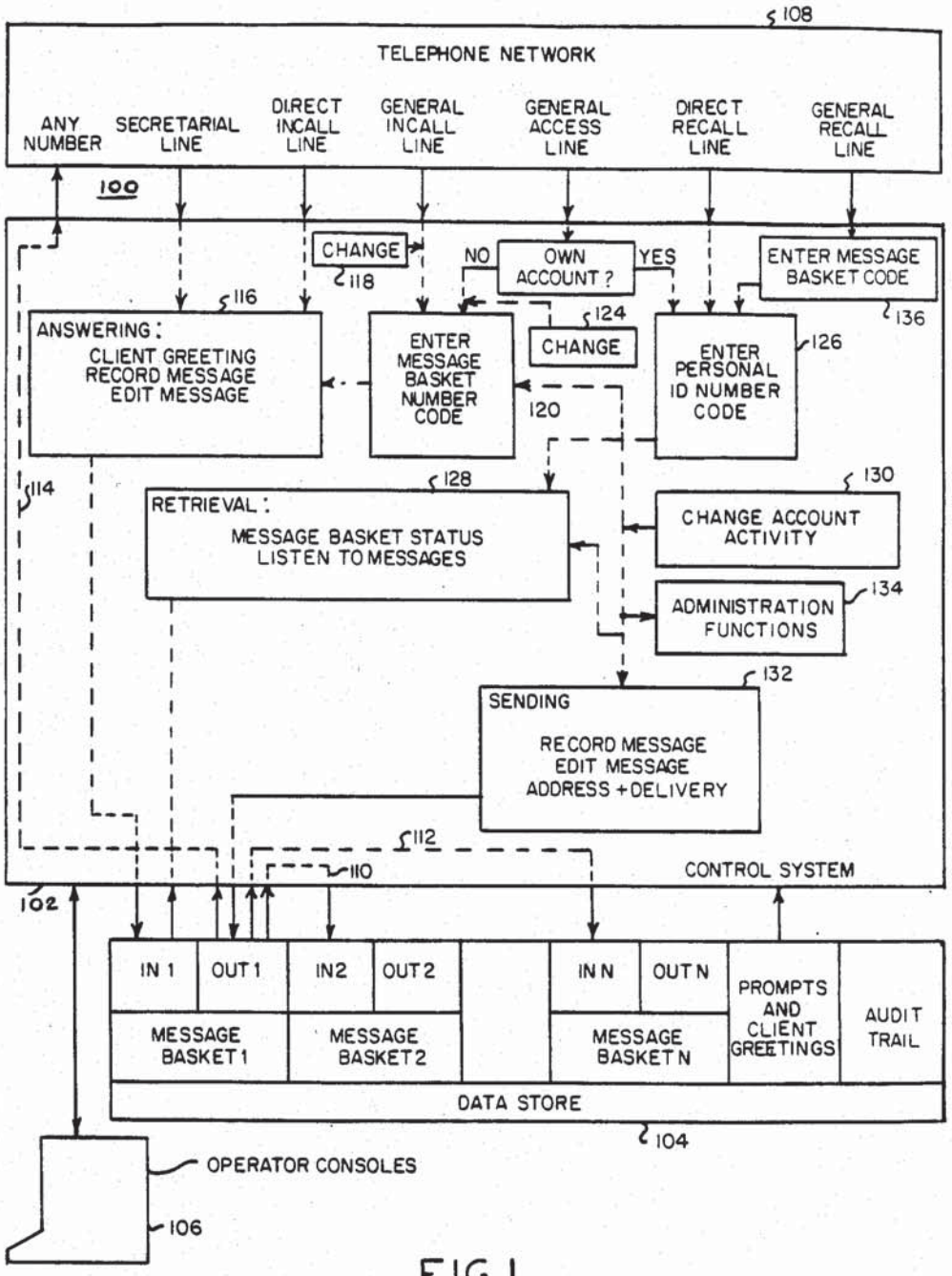
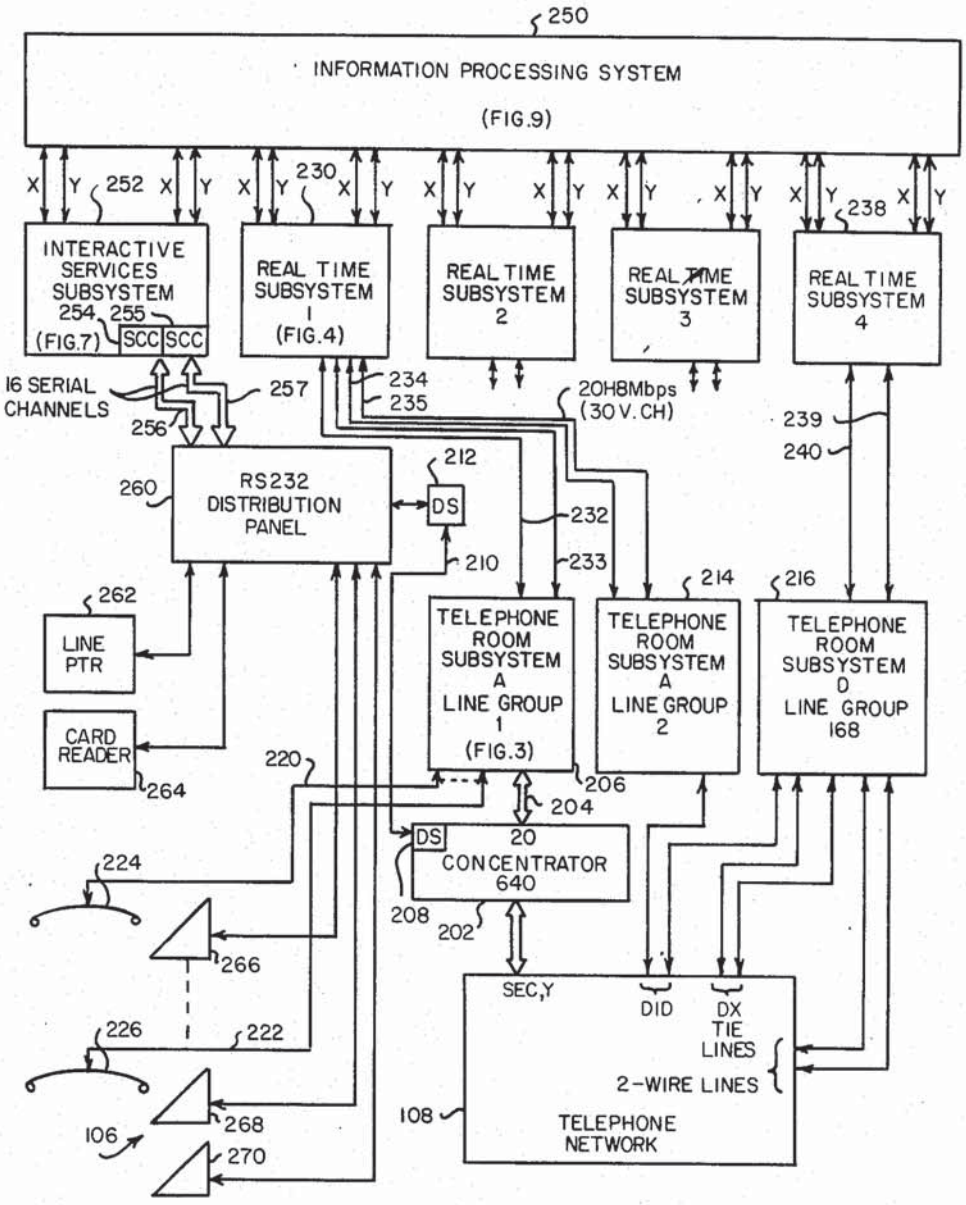
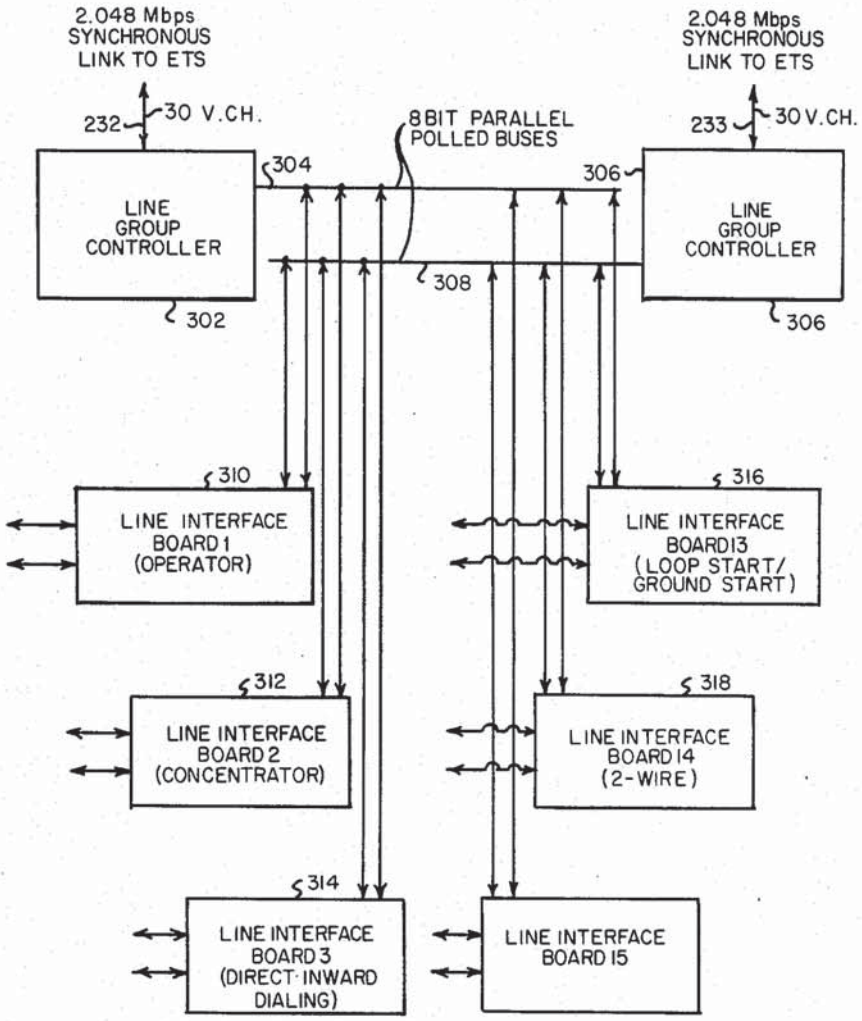

108
TELEPHONE NETWORK
ANY NUMBER
SECRETARIAL LINE
DIRECT INCALL LINE
GENERAL INCALL LINE
GENERAL ACCESS LINE
DIRECT RECALL LINE
GENERAL RECALL LINE
100
CHANGE
118
NO
OWN ACCOUNT ?
YES
ENTER MESSAGE BASKET CODE
136
116
ANSWERING :
CLIENT GREETING
RECORD MESSAGE
EDIT MESSAGE
ENTER MESSAGE BASKET NUMBER CODE
124
CHANGE
ENTER PERSONAL ID NUMBER CODE
126
120
114
128
RETRIEVAL :
MESSAGE BASKET STATUS
LISTEN TO MESSAGES
130
CHANGE ACCOUNT ACTIVITY
ADMINISTRATION FUNCTIONS
134
132
SENDING
RECORD MESSAGE
EDIT MESSAGE
ADDRESS + DELIVERY
112
110
CONTROL SYSTEM
102
IN 1
OUT 1
IN2
OUT2
IN N
OUT N
PROMPTS AND CLIENT GREETINGS
AUDIT TRAIL
MESSAGE BASKET 1
MESSAGE BASKET 2
MESSAGE BASKET N
DATA STORE
104
OPERATOR CONSOLES
106

FIG. 1



TELEPHONE VOICE SERVICE SYSTEM 100

FIG.2



TELEPHONE ROOM SUBSYSTEM A, LINE GROUP 1 206

FIG. 3

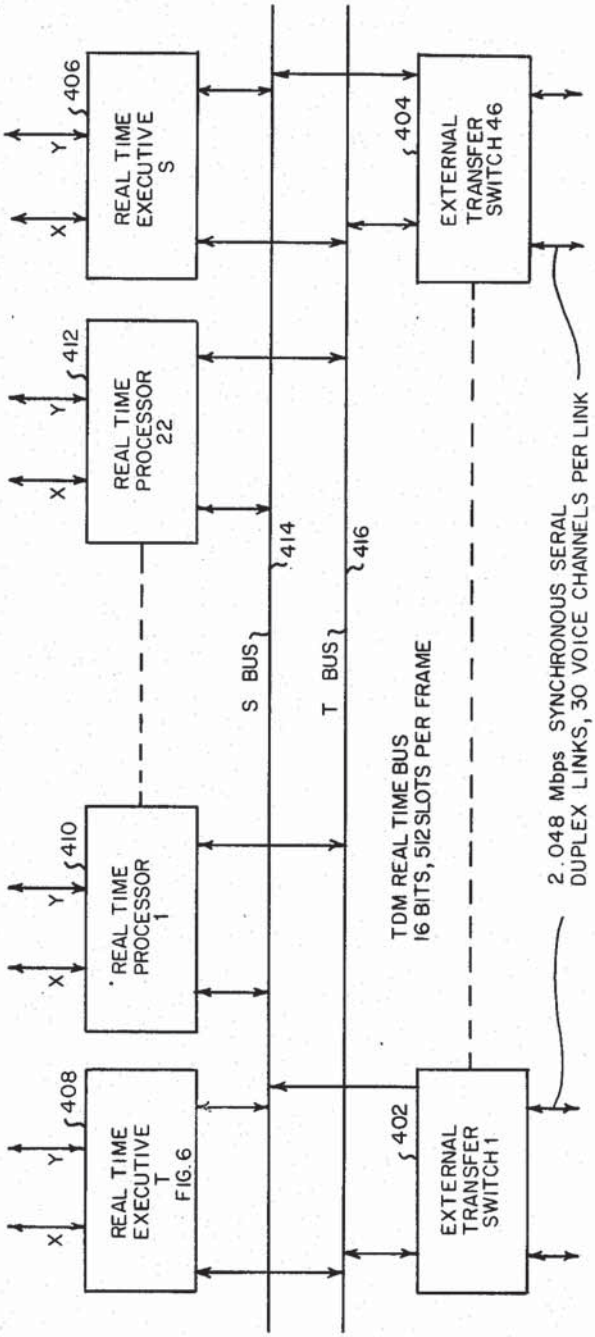

REAL TIME EXECUTIVE S
406
REAL TIME PROCESSOR 22
412
REAL TIME PROCESSOR 1
410
REAL TIME EXECUTIVE T FIG. 6
408
EXTERNAL TRANSFER SWITCH 46
404
EXTERNAL TRANSFER SWITCH 1
402
S BUS
414
T BUS
416
TDM REAL TIME BUS
16 BITS, 512 SLOTS PER FRAME
2.048 Mbps SYNCHRONOUS SERAL
DUPLEX LINKS, 30 VOICE CHANNELS PER LINK
REAL TIME SUBSYSTEM 230

FIG.4

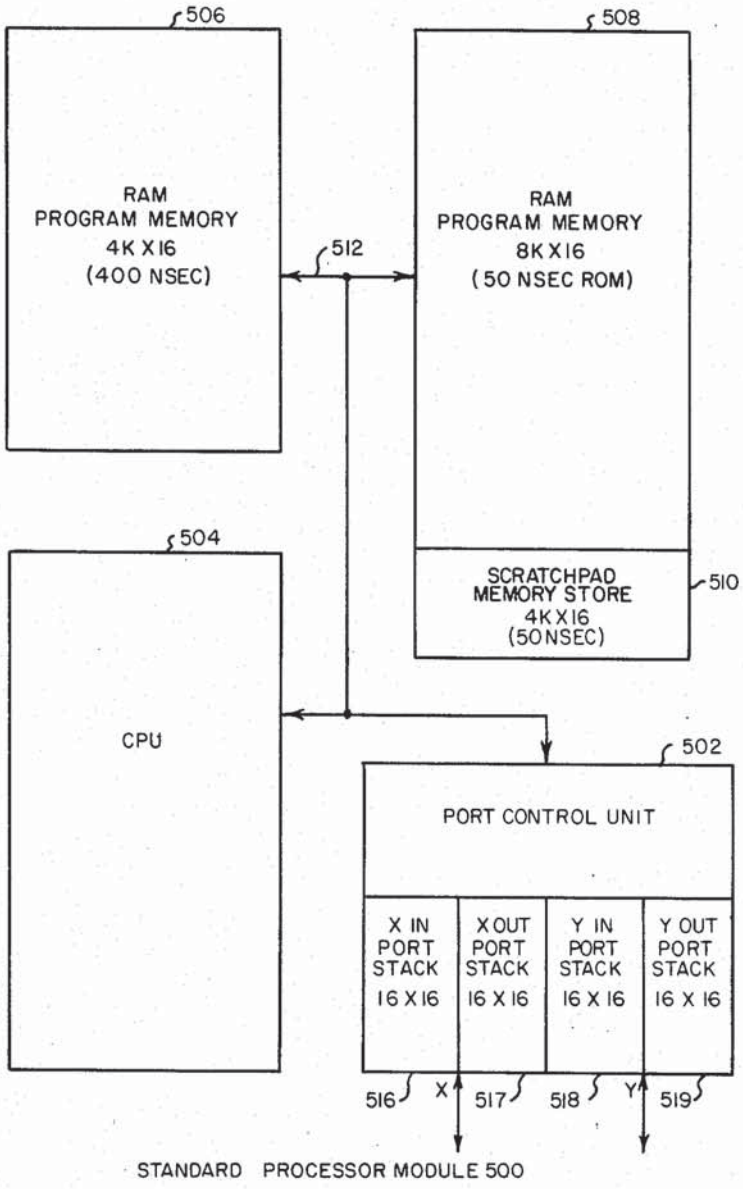

506
RAM
PROGRAM MEMORY
4K X 16
(400 NSEC)
512
508
RAM
PROGRAM MEMORY
8K X 16
(50 NSEC ROM)
SCRATCHPAD
MEMORY STORE
4K X 16
(50 NSEC)
510
504
CPU
502
PORT CONTROL UNIT
X IN PORT STACK 16 X 16
X OUT PORT STACK 16 X 16
Y IN PORT STACK 16 X 16
Y OUT PORT STACK 16 X 16
516
X
517
518
Y
519
STANDARD PROCESSOR MODULE 500

FIG. 5

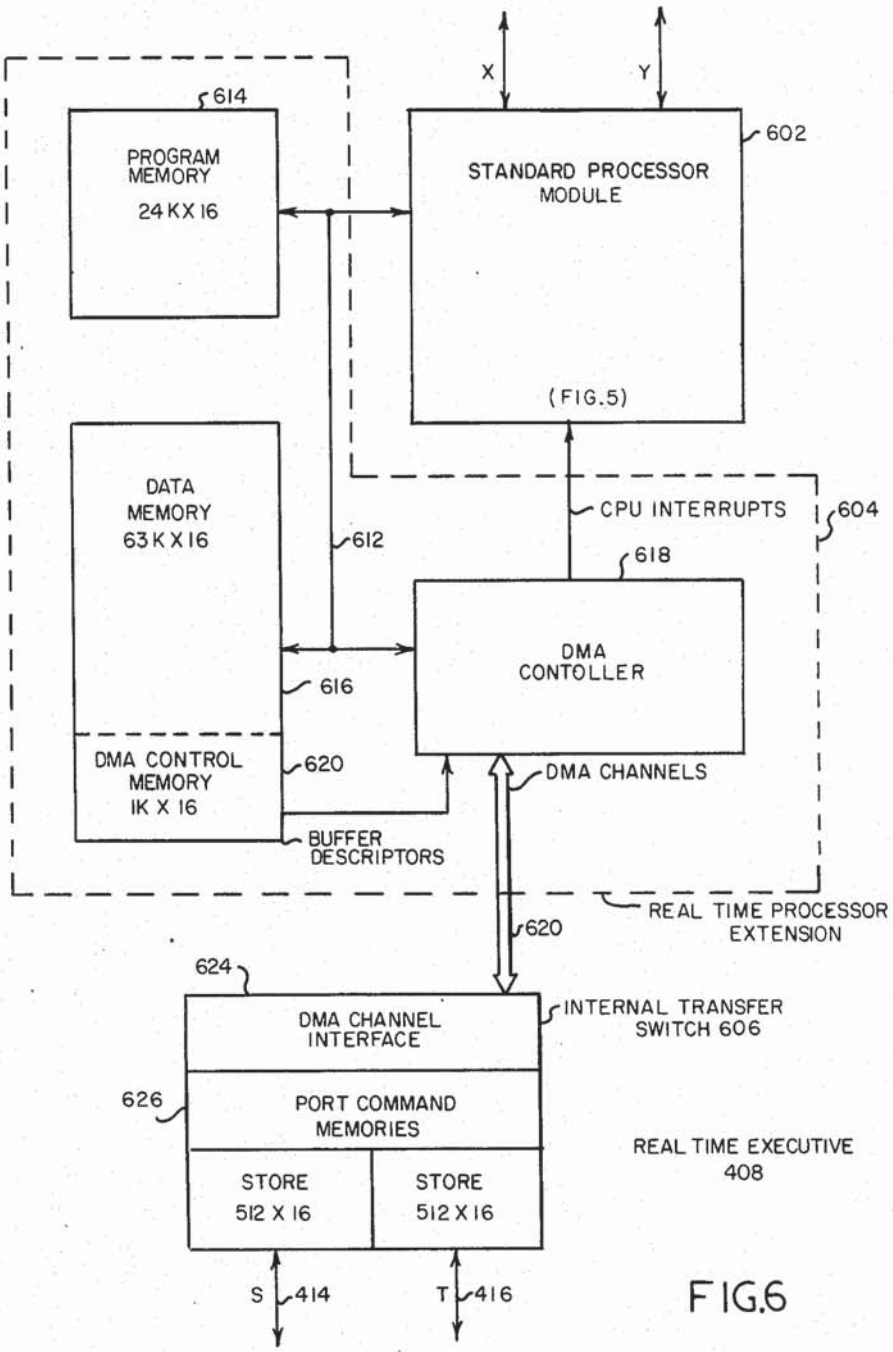

X
Y
614
PROGRAM MEMORY
24 KX 16
602
STANDARD PROCESSOR MODULE
(FIG.5)
DATA MEMORY
63 K X 16
612
CPU INTERRUPTS
604
618
DMA CONTOLLER
616
DMA CONTROL MEMORY
IK X 16
620
DMA CHANNELS
BUFFER DESCRIPTORS
REAL TIME PROCESSOR EXTENSION
620
624
DMA CHANNEL INTERFACE
INTERNAL TRANSFER SWITCH 606
626
PORT COMMAND MEMORIES
REAL TIME EXECUTIVE 408
STORE 512 X 16
STORE 512 X 16
S
414
T
416

FIG.6

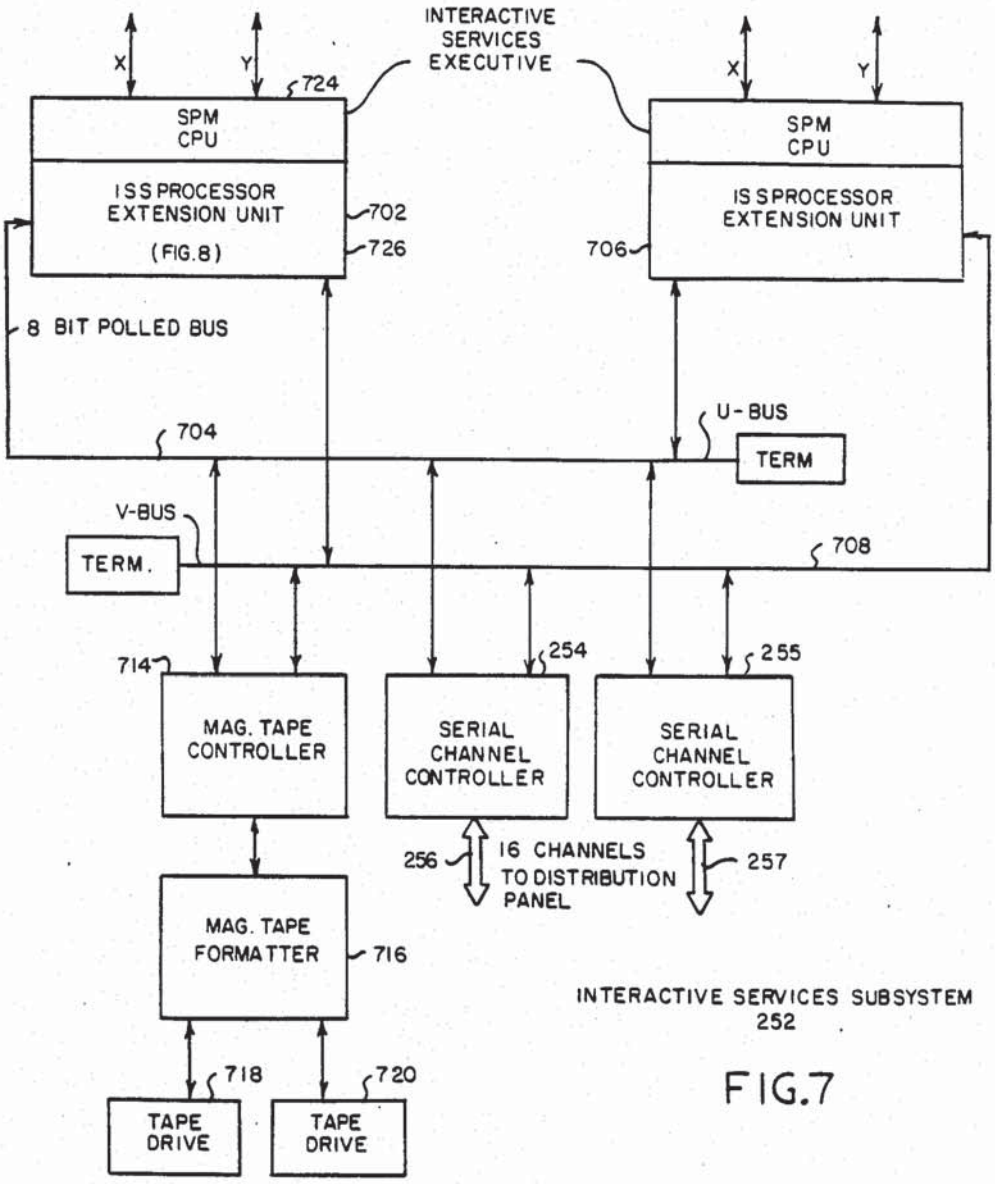

INTERACTIVE SERVICES EXECUTIVE
X
Y
724
SPM CPU
ISS PROCESSOR EXTENSION UNIT
(FIG.8)
702
726
706
8 BIT POLLED BUS
704
U-BUS
TERM
V-BUS
TERM.
708
714
254
255
MAG. TAPE CONTROLLER
SERIAL CHANNEL CONTROLLER
256
16 CHANNELS TO DISTRIBUTION PANEL
257
MAG. TAPE FORMATTER
716
718
720
TAPE DRIVE
INTERACTIVE SERVICES SUBSYSTEM 252

FIG.7

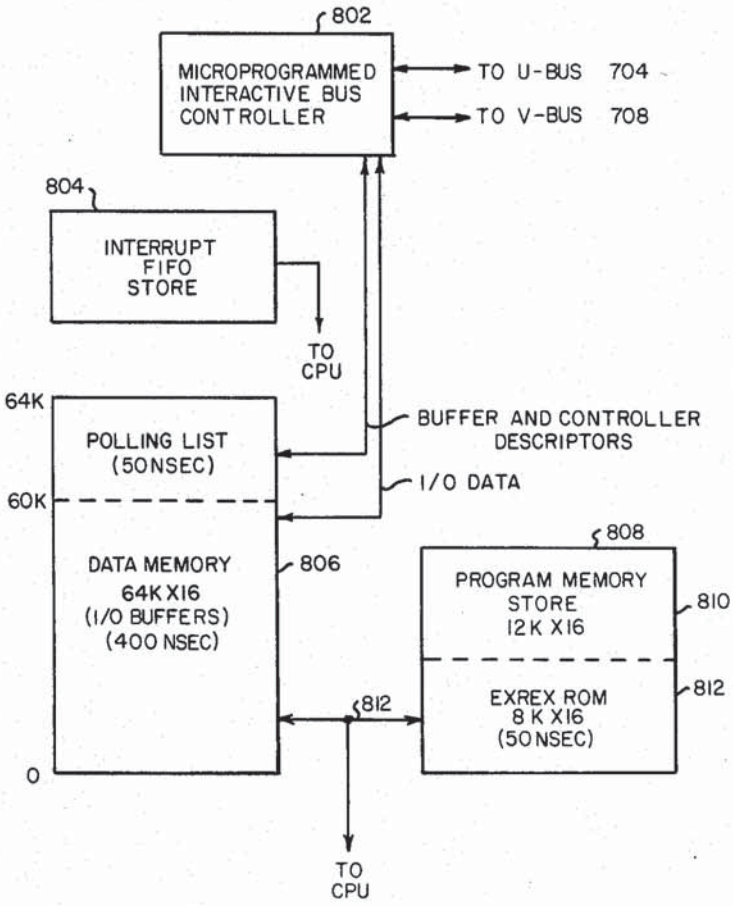

802
MICROPROGRAMMED INTERACTIVE BUS CONTROLLER
TO U-BUS 704
TO V-BUS 708
804
INTERRUPT FIFO STORE
TO CPU
64K
POLLING LIST (50NSEC)
60K
BUFFER AND CONTROLLER DESCRIPTORS
I/O DATA
808
DATA MEMORY 64K X16 (I/O BUFFERS) (400 NSEC)
806
PROGRAM MEMORY STORE 12K X16
810
812
EXREX ROM 8K X16 (50NSEC)
812
0
TO CPU
ISS PROCESSOR EXTENSION UNIT 726

FIG. 8

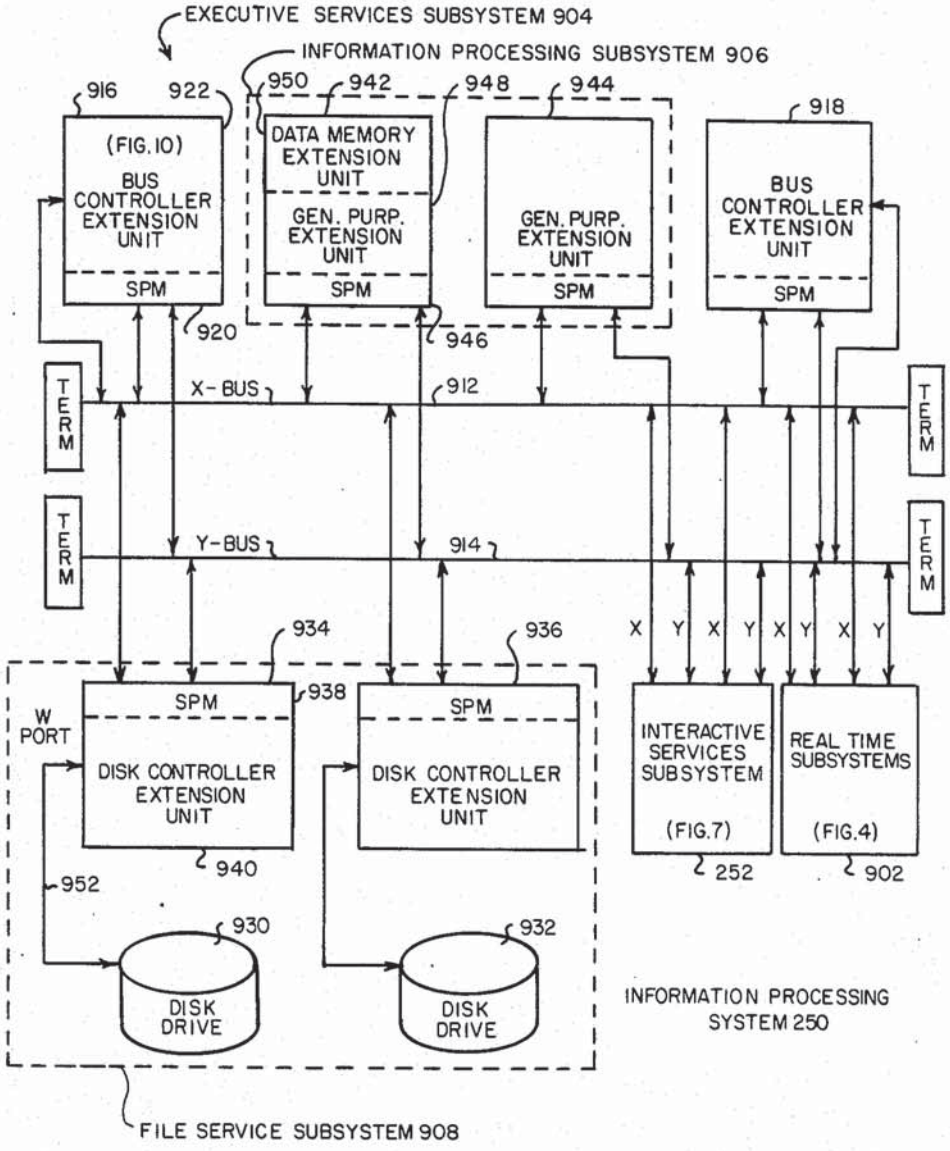

EXECUTIVE SERVICES SUBSYSTEM 904
INFORMATION PROCESSING SUBSYSTEM 906
916
922
950
942
948
944
918
(FIG. 10)
BUS CONTROLLER EXTENSION UNIT
SPM
DATA MEMORY EXTENSION UNIT
GEN. PURP. EXTENSION UNIT
SPM
GEN. PURP. EXTENSION UNIT
SPM
BUS CONTROLLER EXTENSION UNIT
SPM
920
946
TERM
X-BUS
912
TERM
TERM
Y-BUS
914
TERM
934
936
X Y X Y X Y X Y
W PORT
SPM
938
DISK CONTROLLER EXTENSION UNIT
940
SPM
DISK CONTROLLER EXTENSION UNIT
INTERACTIVE SERVICES SUBSYSTEM
(FIG. 7)
252
REAL TIME SUBSYSTEMS
(FIG. 4)
902
952
930
932
DISK DRIVE
DISK DRIVE
INFORMATION PROCESSING SYSTEM 250
FILE SERVICE SUBSYSTEM 908

FIG. 9

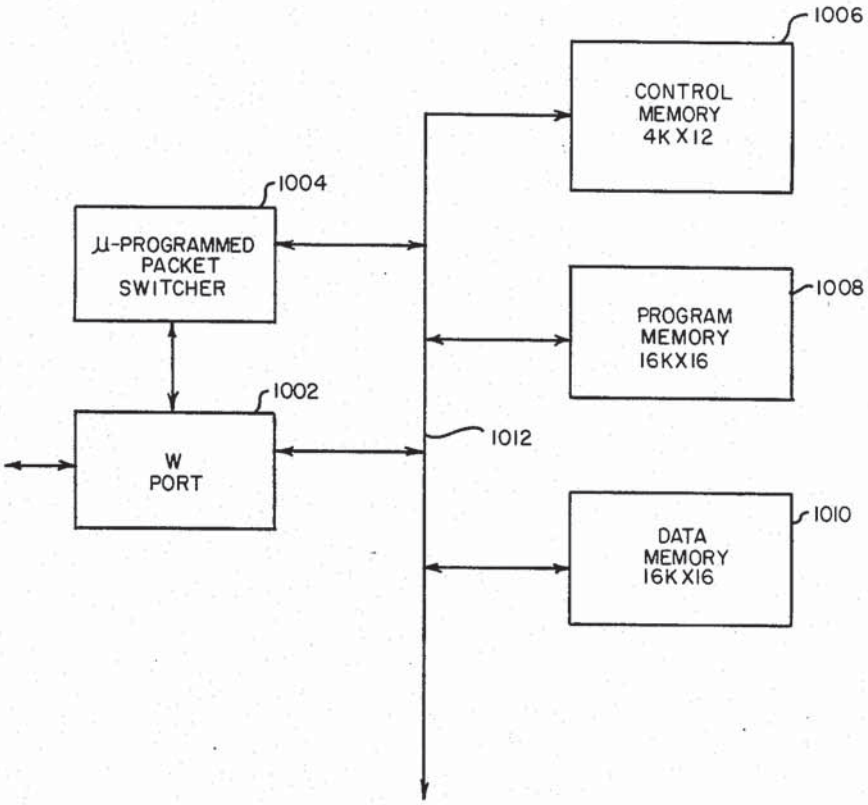

1006
CONTROL MEMORY 4K X 12
1004
μ-PROGRAMMED PACKET SWITCHER
1008
PROGRAM MEMORY 16KX16
1002
1012
W PORT
1010
DATA MEMORY 16K X16
BUS CONTROLLER EXTENSION 922

FIG. 10

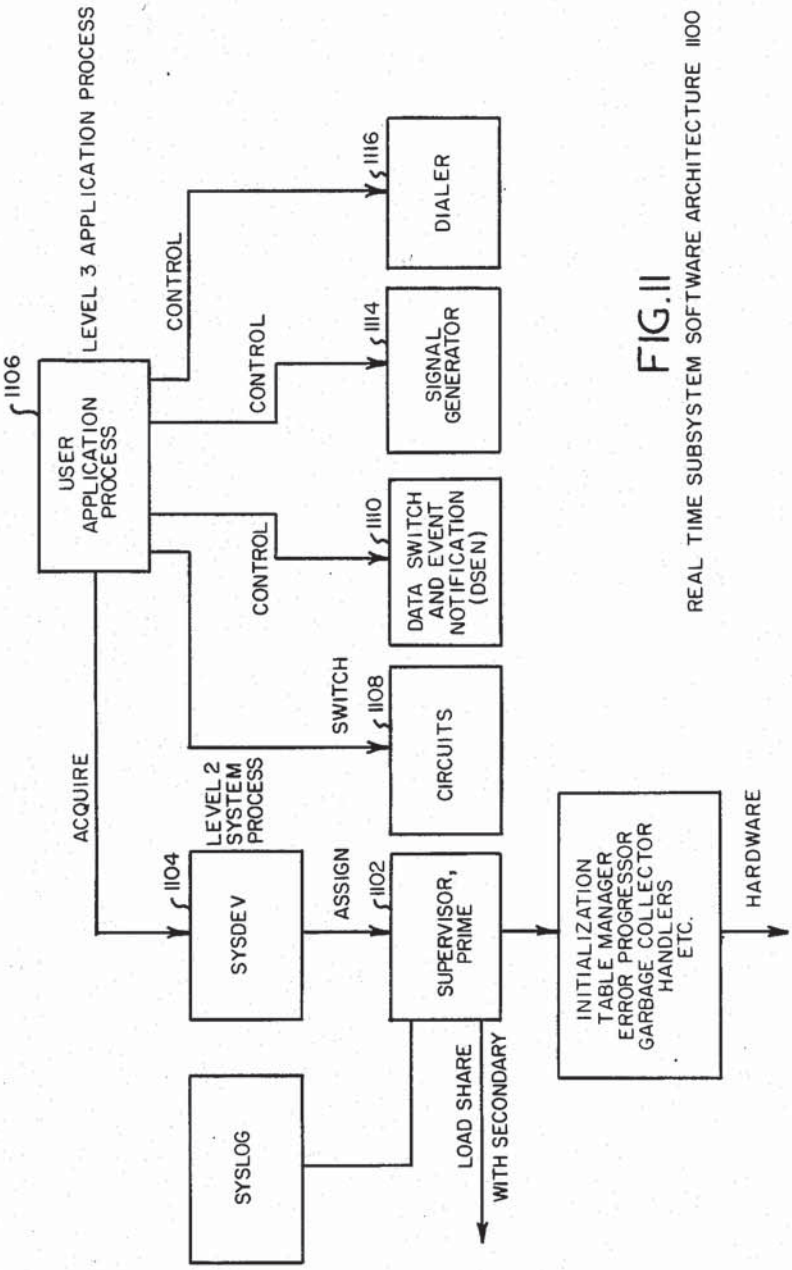
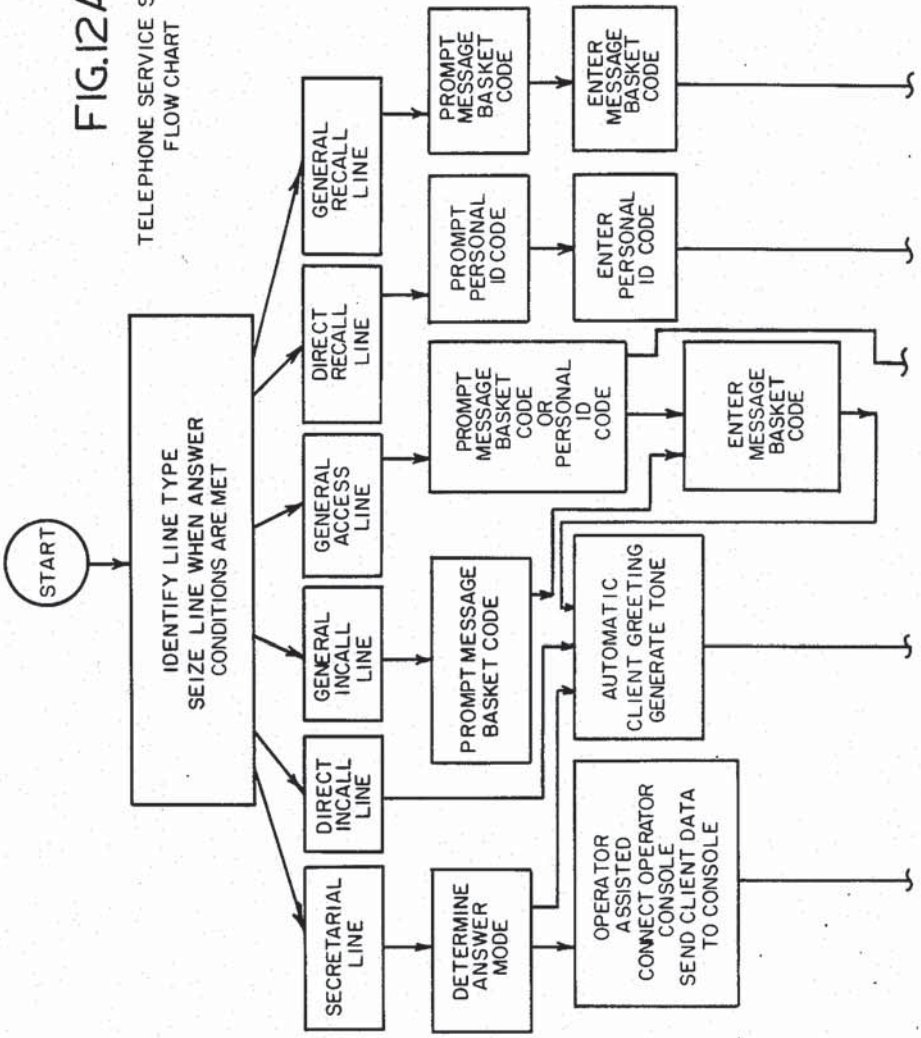


FIG. 11

REAL TIME SUBSYSTEM SOFTWARE ARCHITECTURE 1100

FIG. 12A

TELEPHONE SERVICE SYSTEM
FLOW CHART



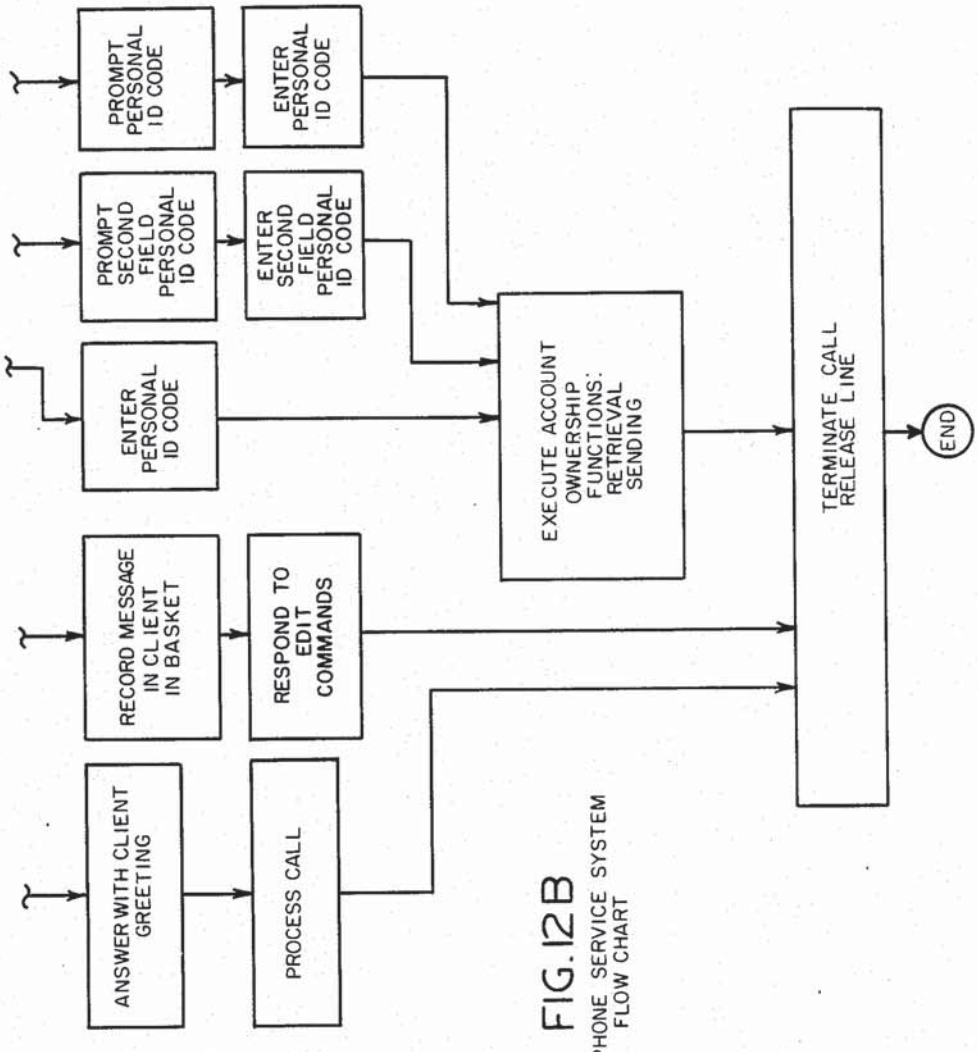


FIG. 12B

TELEPHONE SERVICE SYSTEM FLOW CHART

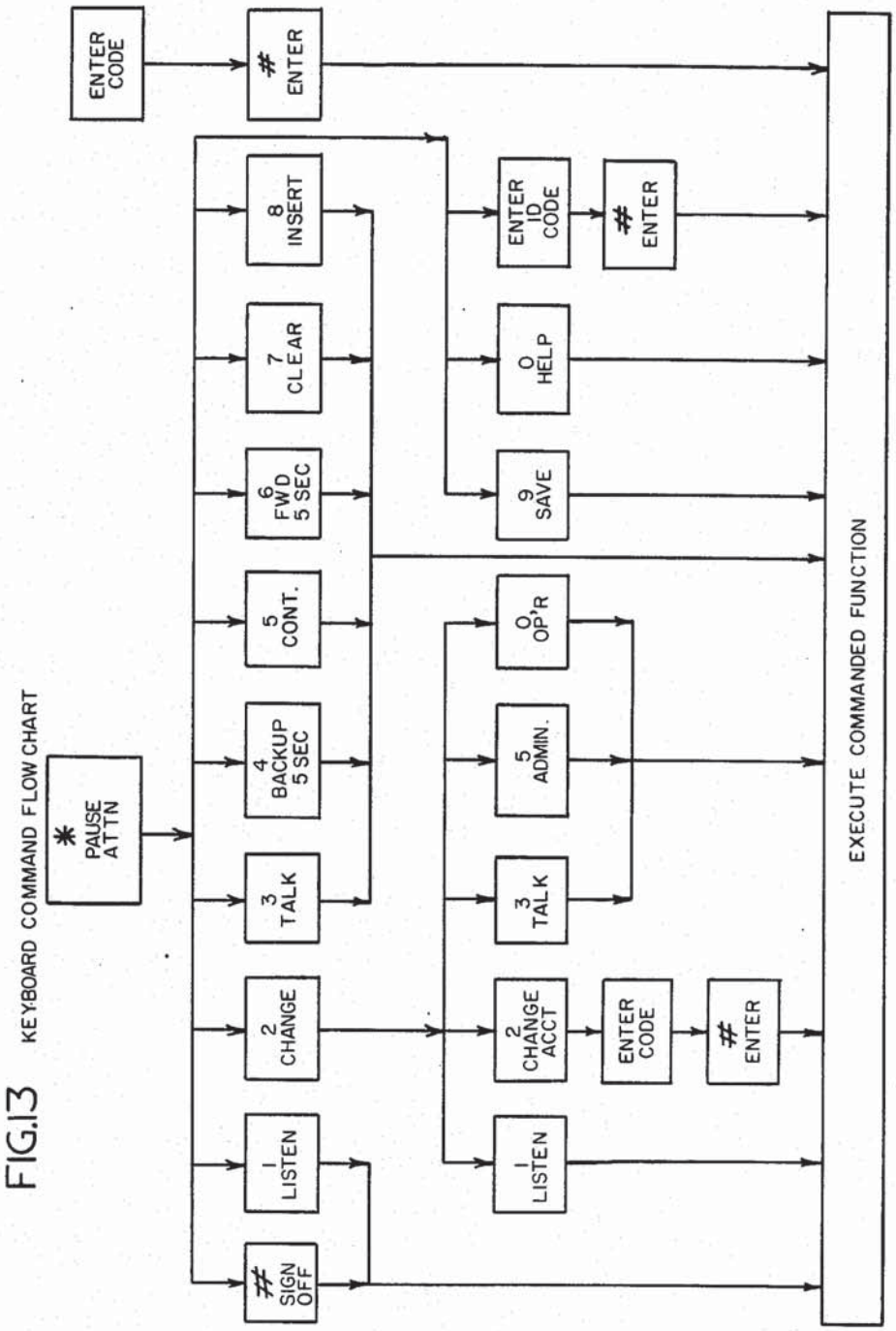

KEYBOARD COMMAND FLOW CHART
* PAUSE ATTN
1 LISTEN
2 CHANGE
3 TALK
4 BACKUP 5 SEC
5 CONT.
6 FWD 5 SEC
7 CLEAR
8 INSERT
SIGN OFF
ENTER CODE
ENTER
ENTER ID CODE
ENTER
0 HELP
9 SAVE
0 OP'R
5 ADMIN.
3 TALK
2 CHANGE ACCT
ENTER CODE
ENTER
1 LISTEN
EXECUTE COMMANDED FUNCTION

FIG.13

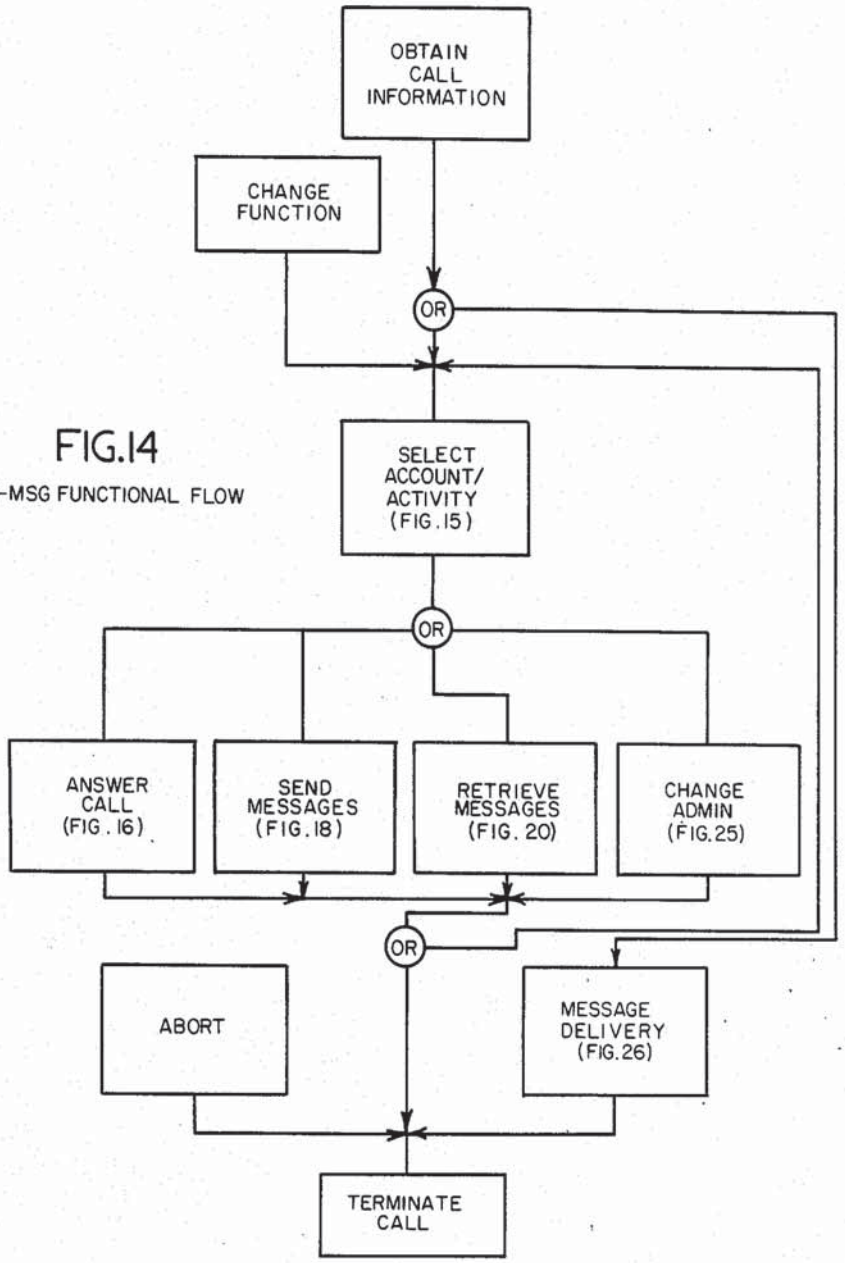
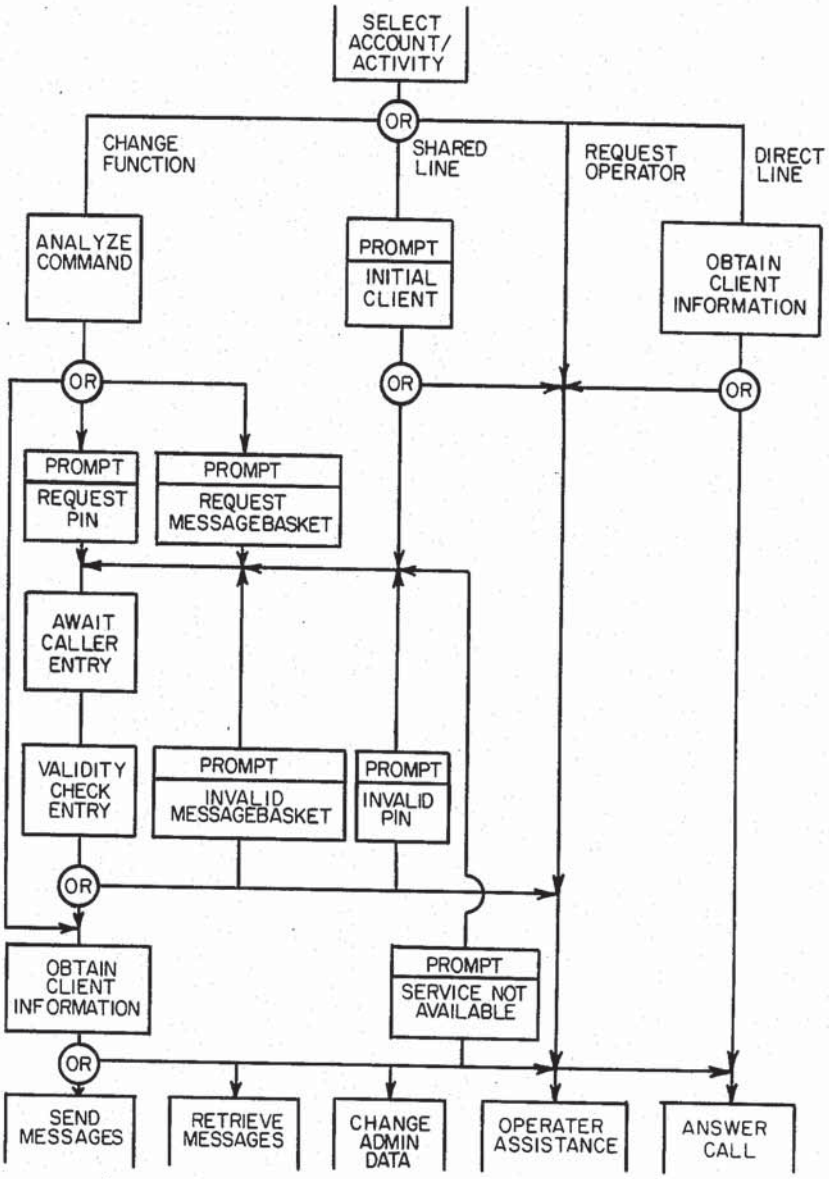


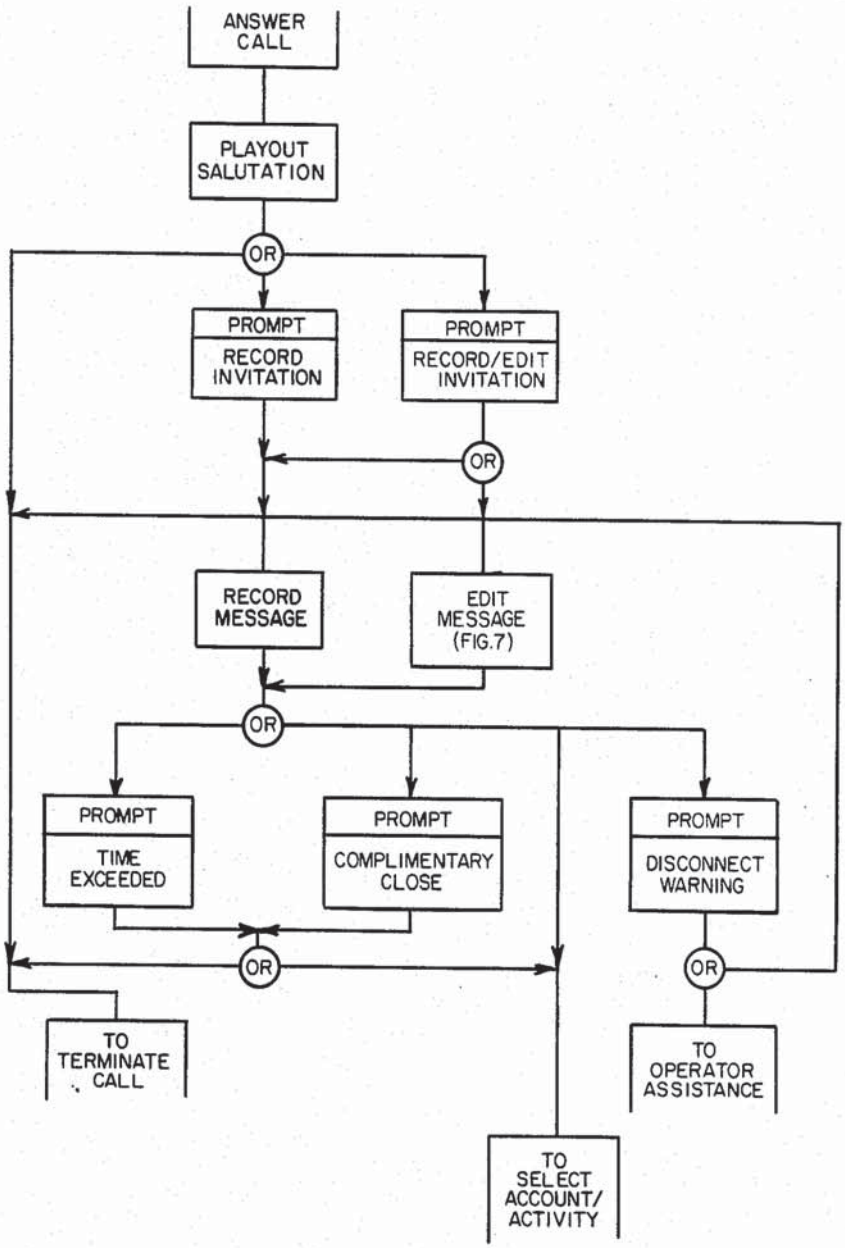
FIG. 14
V-MSG FUNCTIONAL FLOW

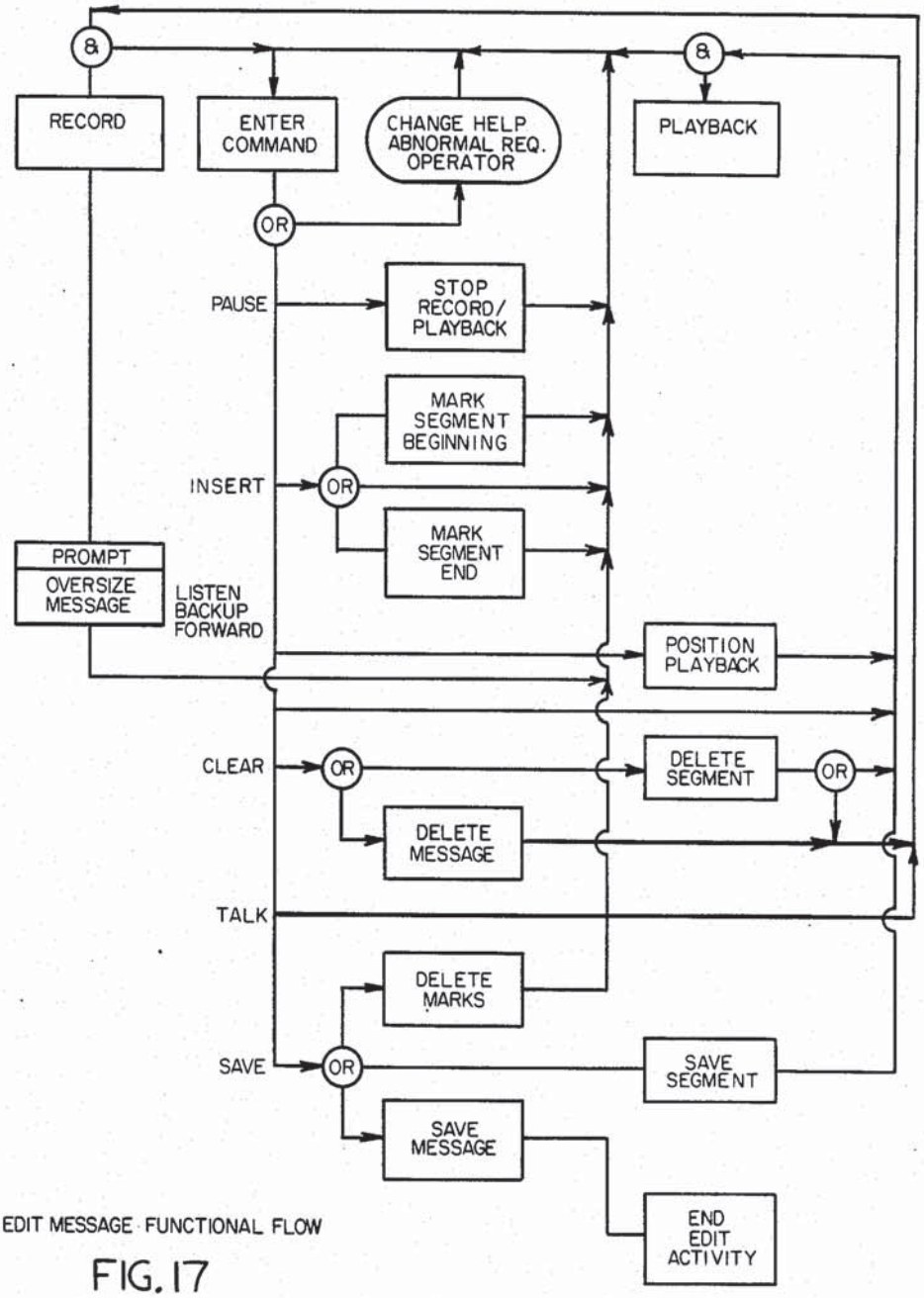


SELECT ACTIVITY FUNCTIONAL FLOW

FIG. 15

FIG.16 ANSWER CALL FUNCTIONAL FLOW





EDIT MESSAGE FUNCTIONAL FLOW

FIG. 17

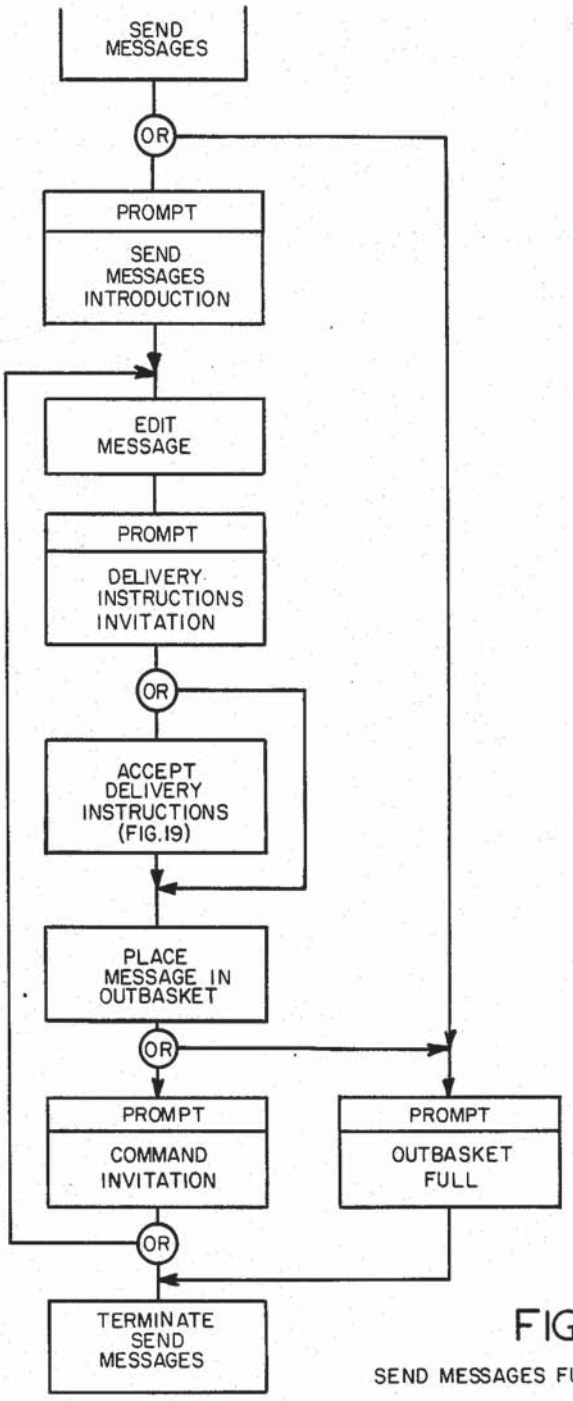
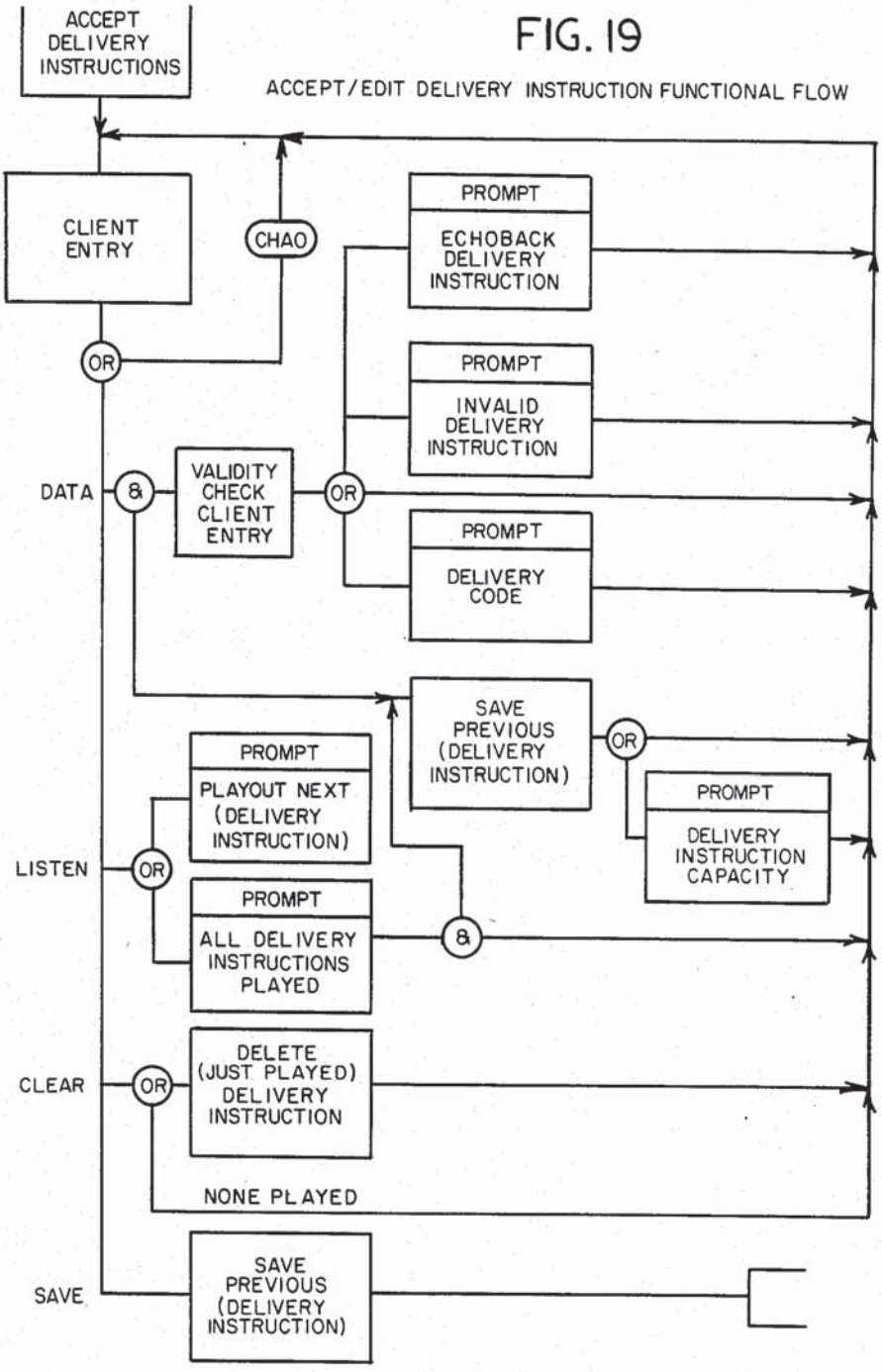


FIG.18

SEND MESSAGES FUNCTIONAL FLOW

FIG. 19

ACCEPT/EDIT DELIVERY INSTRUCTION FUNCTIONAL FLOW



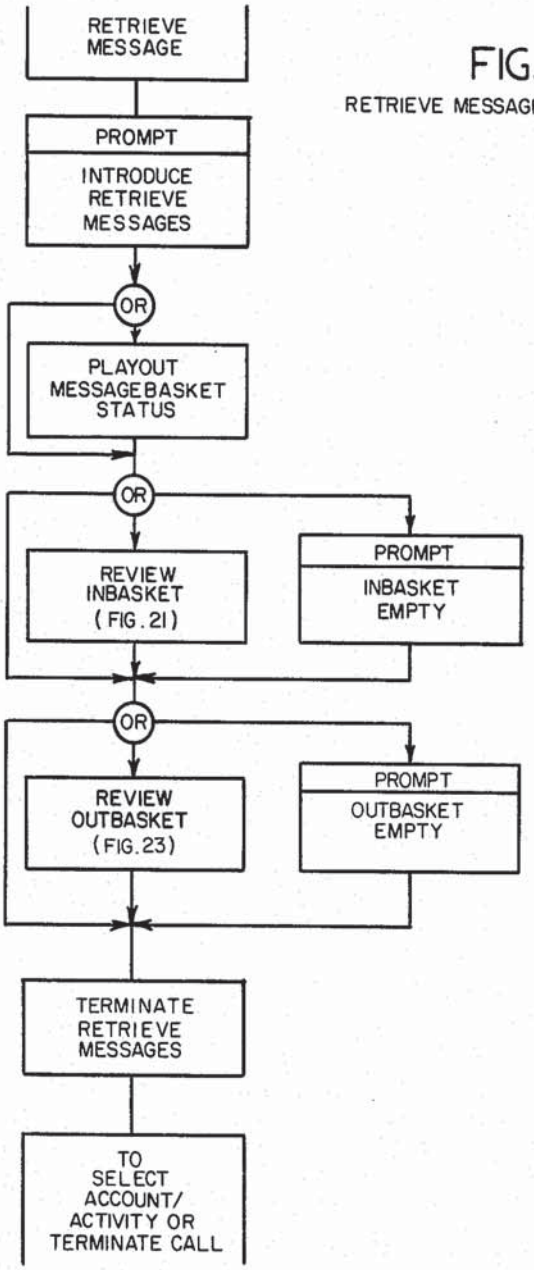
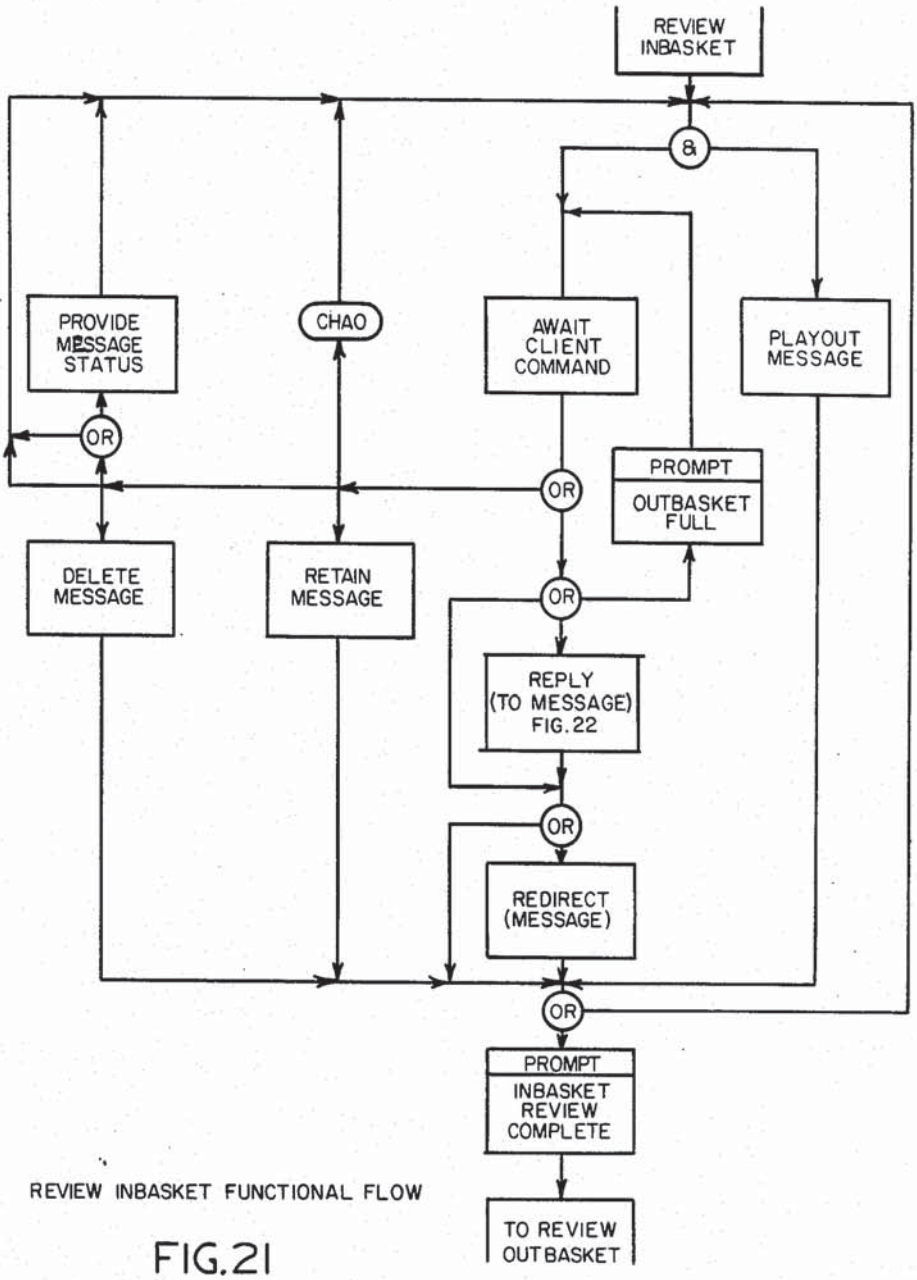


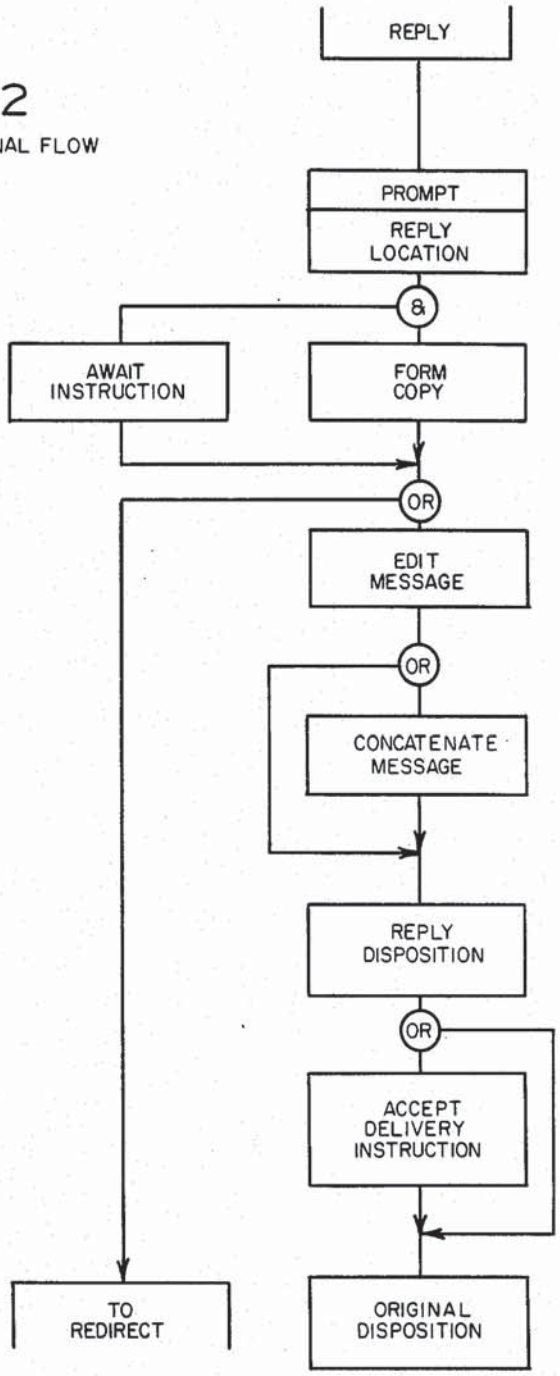
FIG. 20
RETRIEVE MESSAGES FUNCTIONAL FLOW

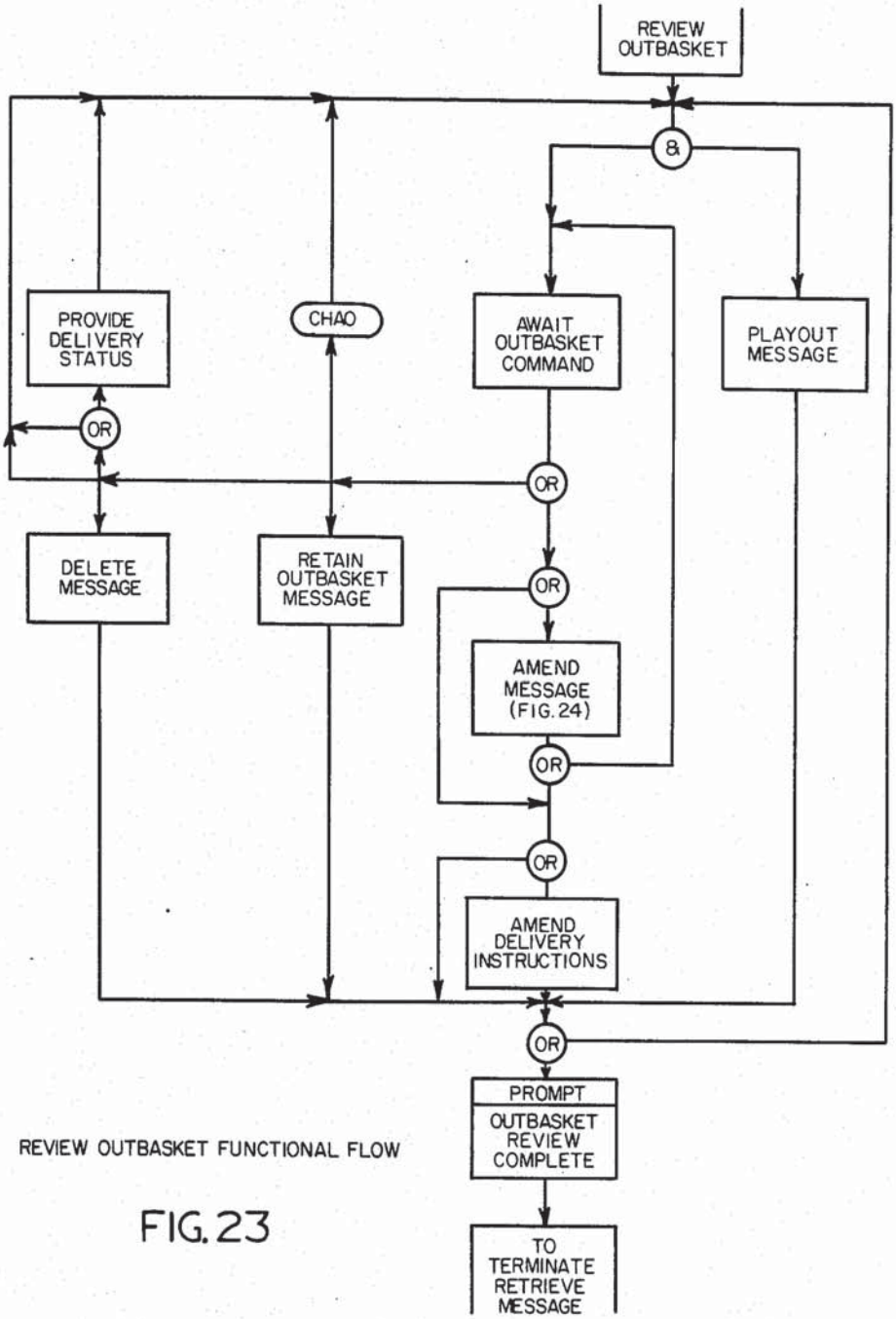


REVIEW INBASKET FUNCTIONAL FLOW

FIG. 21

FIG. 22
REPLY FUNCTIONAL FLOW





REVIEW OUTBASKET FUNCTIONAL FLOW

FIG. 23

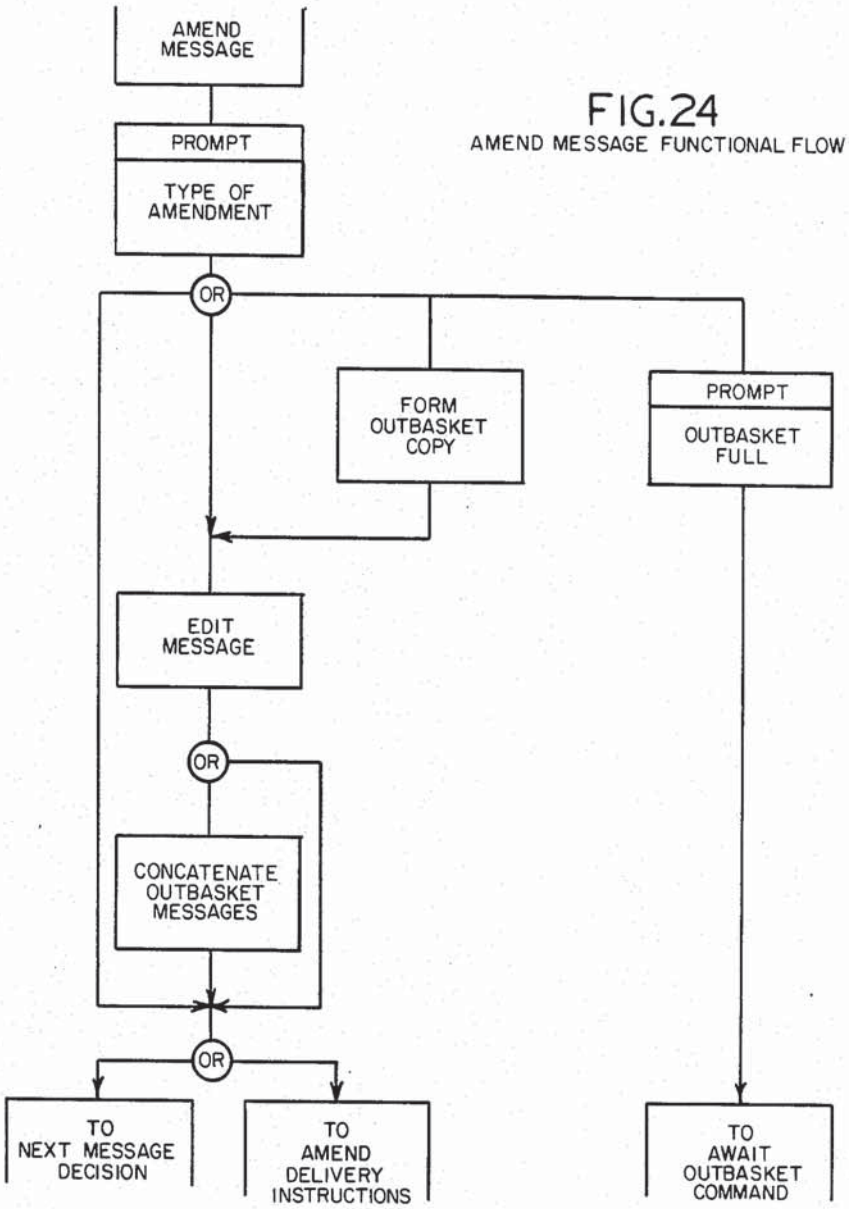


FIG.24
AMEND MESSAGE FUNCTIONAL FLOW

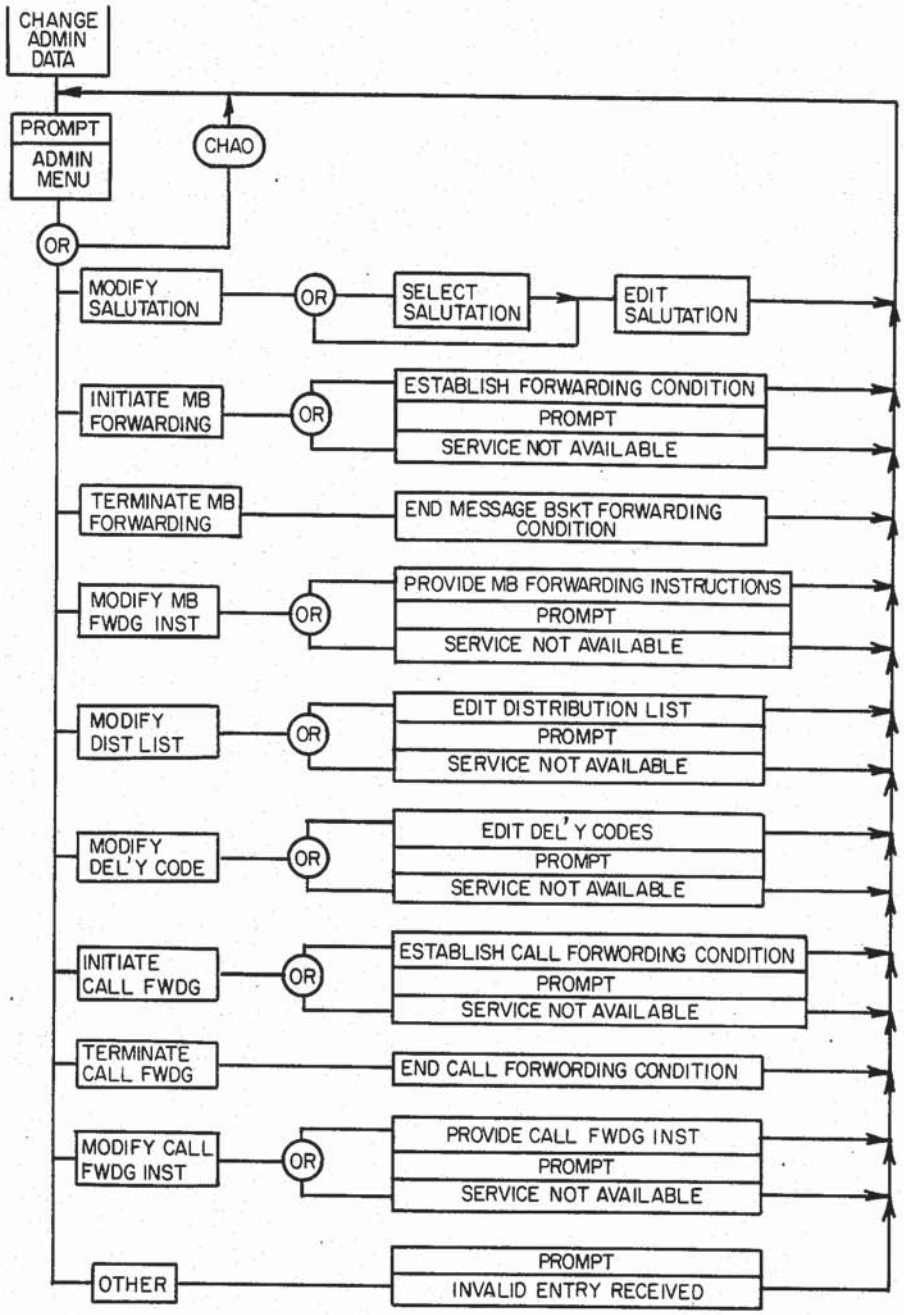


FIG. 25 ADMIN FUNCTIONAL FLOW

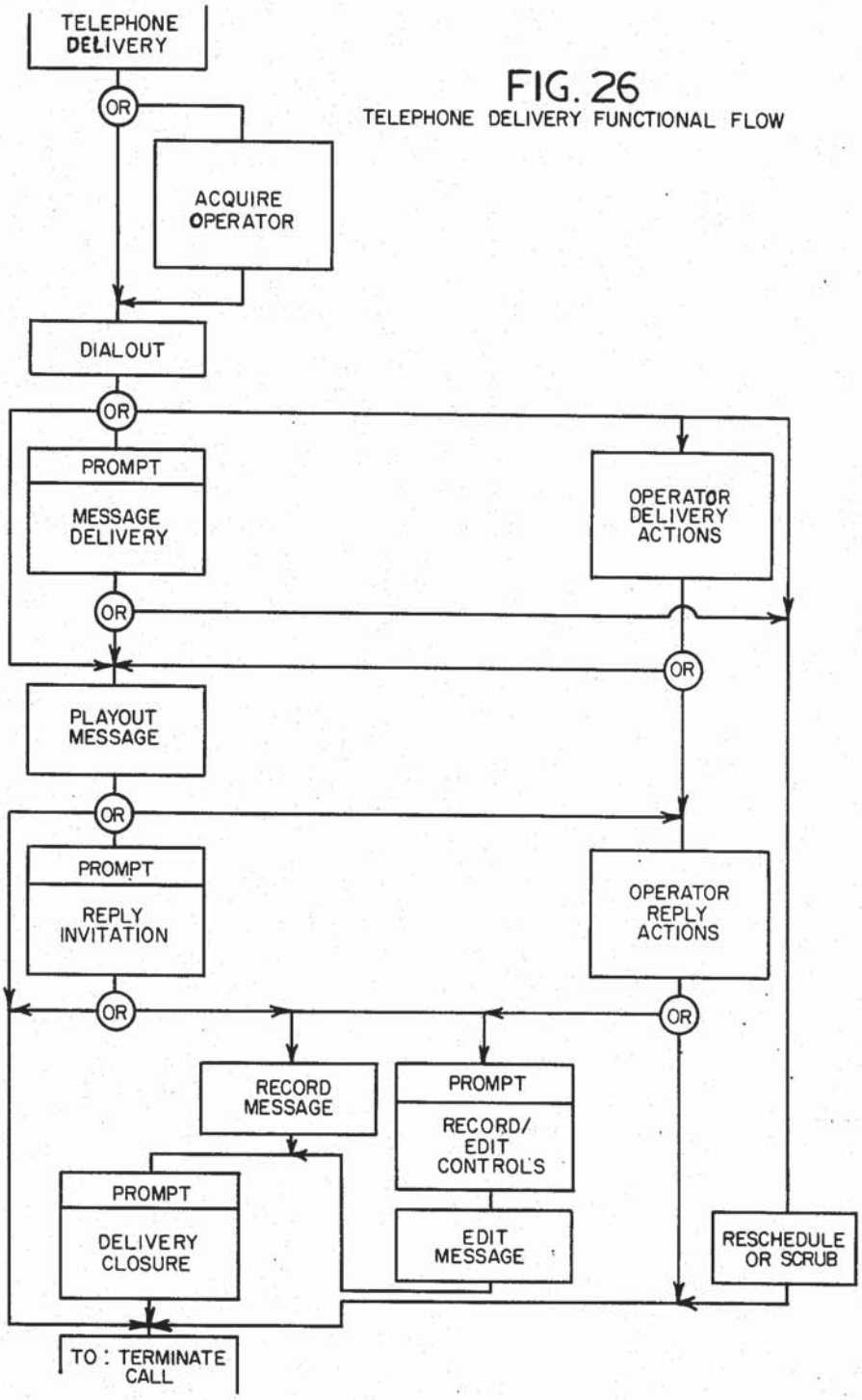


FIG. 26
TELEPHONE DELIVERY FUNCTIONAL FLOW

AUTOMATED TELEPHONE VOICE SERVICE SYSTEM

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to an automated telephone voice service system and more particularly to such a system which provides automatic recording and editing of voice messages as well as forwarding of recorded voice messages to other accounts and telephone numbers with or without operator assistance.

2. Discussion of the Prior Art

Voice responsive telephone service systems have traditionally meant either a telephone answering service or a forwarding service. Early telephone answering service systems connected individual secretarial lines to an operator plug board. Upon activation of one of 100 or more lines coming into an operator station an operator answered the call by making an appropriate plug connection between the active line and an operator headset. Upon identification of an active line the operator could access a corresponding client file to obtain a greeting for reading to the caller. The operator could then proceed to answer questions from information from the client file information or take a handwritten message for storage in a client pidgeon hole until the client called in to retrieve his messages.

Telephone answering systems have subsequently been improved by providing the service operator with a semi-automated terminal to which calls are automatically directed. The line to be answered is automatically identified and corresponding client data is presented to a visual display for use by the operator in answering the call. Any received messages may be keyed into the system for storage in association with the client's account until the client calls to retrieve his messages.

While such a system affords considerable improvement over the early plugboard answering systems, it remains limited to a basic telephone answering mode in which limited client information may be presented to a client and messages may be manually recorded for later retrieval by the client.

The forwarding services provide a somewhat different but still limited telephone service function. These services include store and forward services and call forwarding services. The store and forward services enable a client to record a message and designate a number of persons or telephone numbers for delivery of the message. Delivery instructions can specify dates and times for message delivery. The operator then proceeds to call the indicated persons or numbers in accordance with the delivery instructions and upon completing a call, play the prerecorded message. Such a system permits recording of a message at one time for delivery at another time, but still requires operator assistance.

Call forwarding on the other hand can be completely automated, but merely forwards an incoming call to a previously designated telephone line. Such a system cannot process the call if either the originally called line or the forwarded line are busy and cannot process a message at different times which are respectively convenient to the calling and called parties.

SUMMARY OF THE INVENTION

An automated telephone voice service system in accordance with the invention includes a data store coupled to store and retrieve voice messages at each of a plurality of individually addressable message baskets therein and a control system providing a selective coupling between the store and each of a plurality of telephone lines of a telephone network. The control system is responsive to different data signals received over a particular one of the telephone lines to associate the particular telephone line with a particular message basket, to store in the particular message basket a voice message received over the particular telephone line, to forward a voice message stored in the particular message basket to at least one other of the individually addressable message baskets, and to forward a voice message stored in the particular message basket to at least one telephone line.

The service system is implemented with a high reliability fail soft data processing system in which duplication of processing system components enables a function of a failed component to be transferred to another component to assure that no single failure disables the entire system. The major processor subsystems include a real-time subsystem providing interactive coupling to the analog telephone lines, an interactive services subsystem providing a coupling to input-output devices such as terminals, tape drives, and printers, a file services subsystem hosting a plurality of disk drives, an information processing subsystem providing a sophisticated general multiprocessor capability and an executive services subsystem providing communication and coordination between the other subsystems.

The real time subsystem provides the signal generating, signal detection and interface circuitry necessary for connection to several different physical and functional types of standard network telephone lines. The physically different types of lines include, two-wire lines, four-wire lines, pay telephone lines, operator lines and concentrator lines. The functionally different lines include secretarial lines which are usually coupled through a concentrator and function as jumped extensions of client telephone lines, and direct inward dial lines having virtual connections between an incoming line at a telephone switching office and a called telephone at a customer location (i.e. the telephone voice service system). Plural trunking connections to the service system concentrator and to the direct inward dial lines enable multiple calls to the same number to be processed simultaneously by the service system. Instead of a second or third caller to a given phone number receiving an irritating busy signal, the caller receives a prompt and efficient automatic response.

At the service system particular lines may be internally assigned predetermined designated functions. For example, some of the lines may be designated direct incall lines with each being assigned a predetermined association with a particular message basket. Such a line is controlled only in a telephone answering mode with a caller being greeted by a client selected voice message which may be in the client's own voice and changed at will and then invited to leave a voice message in the associated message basket upon the generation of a tone signal. Callers familiar with the system may edit the voice message using Touch Tone commands, but no editing prompts are provided. There is thus no confusion of callers who are unfamiliar with the system and who wish to simply record an unedited voice message upon the occurrence of the tone.

General incall lines are operated in a telephone answering mode in a manner similar to the direct incall

lines except that the general incall lines do not have a unique, predetermined association with a particular message basket. A caller is prompted to enter through the telephone keyboard dual tone multiple frequency (DTMF) data signals defining commands which select a particular message basket code or address. A voice communication coupling between the calling telephone line and the selected particular message basket is then created by the telephone service system.

Other telephone lines may be assigned as direct or general recall lines which afford a client access to account ownership functions afforded by the system. As with the incall lines each direct recall line is associated with a single predetermined message basket while a general recall lines requires entry of a message basket code identifying a desired message basket. Security is maintained by enabling account ownership activities only after a personal identification code has been entered which corresponds to an associated message basket. Added security may be implemented for a direct recall line by requiring entry of a second field of a personal identification code before account ownership activities are enabled. The second field is separated from the first field by a number sign key center and may be changed at any time by the account owner. Account entry thus requires a caller to have knowledge of the direct recall telephone phone number, the first field of the personal identification code associated therewith, and if used, the second field of the personal identification code.

Account ownership activities include retrieval of messages, forwarding of messages, and administrative functions such as the recording of a new greeting, the changing of answering criteria for a secretarial line or the changing of the second field of the personal identification code. Each message basket is divided into two parts, an inbasket which stores messages from outside callers and an outbasket which stores messages for forwarding to other inbaskets or telephone lines. Data storage space is conceived by storing only a single copy of an outgoing voice message in the client's outbasket, even if the message is to be sent to many different parties.

If the message is to be sent to other message baskets a code is placed in the inbasket of each inbasket portion thereof identifying the particular voice message in the particular outbasket of the sending client. If the message is to be communicated over one or more telephone lines, the outbasket message is simply accessed as the calls are initiated. This arrangement also enables a client to retain ownership of a message so that a message can be changed or deleted until it has actually been delivered.

As a message is delivered to another service client the recipient can direct that the message be stored in the recipient's inbasket for future reference and can automatically direct a voice message reply back to the sender with or without the original message attached.

A general access line affords a caller access to all voice service system functions. Any message basket may be selected for leaving a message therein by entering the message basket number code therefor and entry of a personal identification number code enables access to account ownership activities. To minimize errors and enable the service system to readily distinguish between different types of data sets, different data sets are required to have mutually exclusive code ranges. For example, one digit defines a delivery code selecting a predetermined set of voice message delivery instructions, two digits define a predetermined distribution list, three to nine digits define a message basket number and ten or more digits define a telephone number including the area code even for a local number. A personal identification number code must be preceded by an asterisk (*) and may have any reasonable number of digits within predetermined limits for the system, for example 3–15. All data sets are terminated by an # (enter) key or a 5 second time out.

In the event a system user requires assistance, more detailed voice message prompts are initiated by keying *0 and communication with a voice message operator can be commanded by keying *20. In the event that a client calls the system from a dial telephone, the service system detects a telephone company signal identifying a dial telephone line as the source of the call and automatically connects a service system operator to the line. The telephone service system in accordance with the invention thus provides a sophisticated user controlled system for the receipt and delivery of voice messages with an operator being required only for exceptional circumstances.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the invention may be had from a consideration of the following Detailed Description taken in conjunction with the accompanying drawings in which:

FIG. **1** is a functional block diagram representation of an automated telephone voice service system in accordance with the invention;

FIG. **2** is a block diagram representation of the system architecture for voice service system shown in FIG. **1**;

FIG. **3** is a block diagram representation of a telephone room subsystem line group used in the telephone voice service system shown in FIG. **2**;

FIG. **4** is a block diagram representation of a real time subsystem used in the telephone voice service system shown in FIG. **2**;

FIG. **5** is a block diagram representation of a standard processor module used in the voice service system shown in FIG. **1**;

FIG. **6** is a block diagram representation of a real time executive used in the real time subsystem shown in FIG. **4**;

FIG. **7** is a block diagram representation of an interactive services subsystem used in the telephone voice service system shown in FIG. **2**;

FIG. **8** is a block diagram representation of an interactive services subsystem processor extension unit shown in FIG. **7**;

FIG. **9** is a block diagram representation of an information processing system shown in FIG. **2**;

FIG. **10** is a block diagram representation of a bus controller extension unit shown in FIG. **9**;

FIG. **11** is a block diagram representation of the software architecture for the real time subsystem shown in FIG. **4**;

FIG. **12A** in conjunction with FIG. **12B** is a flow diagram describing the response of the automatic telephone voice service system to a user call;

FIG. **13** is a flow diagram of telephone keyboard command operations.

FIG. **14** is a voice messaging functional flow diagram for the service system shown in FIG. **1**;

FIG. **15** is a select activity functional flow diagram that is useful in understanding the diagram shown in FIG. **14**;

FIG. **16** is an answer call functional flow diagram that is useful in understanding the diagram shown in FIG. **14**;

FIG. **17** is an edit message functional flow diagram that is useful in understanding the diagram shown in FIG. **16**;

FIG. **18** is a send messages functional flow diagram that is useful in understanding the diagram shown in FIG. **14**;

FIG. **19** is an accept/edit delivery instructions functional flow diagram that is useful in understanding the diagram shown in FIG. **18**;

FIG. **20** is a retrieve messages functional flow diagram that is useful in understanding the diagram shown in FIG. **14**;

FIG. **21** is a review inbasket functional flow diagram that is useful in understanding the diagram shown in FIG. **20**;

FIG. **22** is a reply functional flow diagram that is useful in understanding the diagram shown in FIG. **21**;

FIG. **23** is a review outbasket functional flow diagram that is useful in understanding the diagram shown in FIG. **20**;

FIG. **24** is an amend message functional flow diagram that is useful in understanding the diagram shown in FIG. **23**;

FIG. **25** is an administration functional flow diagram that is useful in understanding the diagram shown in FIG. **14**; and

FIG. **26** is a telephone delivery functional flow diagram that is useful in understanding the diagram shown in FIG. **14**.

DETAILED DESCRIPTION GENERAL BACKGROUND

Referring now to FIG. **1**, an automated telephone voice service system **100** according to the invention includes a control system **102** coupling a data store **104** and one or more operator consoles **106** to a standard telephone network **108** which may represent all of the interconnectable telephones throughout the United States and the world. The data store **104** is shown as a single functional block divided into a plurality of addressable units. However, as is conventional the data store **104** may be physically implemented as one or more magnetic or electronic storage devices and may be distributed throughout a data processing system. Data store **104** provides storage for a plurality of addressable message baskets designated message basket **1** through message basket N, a plurality of individually addressable voice message prompts and client greetings, and an audit trail for each client accessing the system **100**.

Each message basket provides storage for a plurality of voice messages and is segregated into an inbasket section and an outbasket section. Each inbasket section stores voice messages and message forwarding notices directed by system users to client owners of the associated message basket. The inbasket of each message basket functions in a manner analogous to a recording mechanism for a telephone answering machine.

The outbasket portion of each message basket receives voice messages generated by the message basket account order for forwarding to selected other message baskets or to telephohe network **108** users at indicated telephone numbers. The forwarding of a message from an outbasket to an inbasket as represented by arrows **110**, **112** is accomplished automatically without human intervention while the forwarding of a message from an outbasket to a telephone network **108** user at a selected telephone number as indicated by arrow **114** may be accomplished either automatically or semiautomatically with operator assistance as required for compliance with the instructions of the client account and applicable state law. For example, in a fully automatic mode, the control system **102** can operate to call the indicated telephone number and upon its being answered, communicate an appropriate recorded voice message prompt, communicate the voice message being sent, and then terminate the call. As an example, the voice message prompt might inform the person answering the telephone at the indicated number that the person is about to receive a prerecorded message from John Doe, the account owner. This mode of operation enables the account owner to record a single message in his outbasket and have the message broadcast to one or thousands of designated recipients without any further effort by the account owner. The account owner, when setting up, or modifying his account, establishes predetermined distribution lists and sets of delivery instructions, each having a different selection code number. The delivery instructions can cover such features as days of the week and time intervals during which delivery may be made, number of retries, and whether the forwarding of the message is to be accomplished automatically or semiautomatically with operator assistance.

In the semiautomatic mode, the control system **102** waits for delivery conditions to be met, and then obtains ownership of an active operator console including a terminal having a keyboard and a video display unit and an operator headset. The control system **102** informs the operator through the console **106** that a semiautomated message forwarding operation is to be undertaken and displays a prompting message for the operator to read. Upon command, the control system **102** generates the Touch Tone signals corresponding to the recipient's telephone number and connects the operator console **106** to the line when it is answered. The operator informs the answering party of the call, asks to talk to a particular person at the called telephone number if appropriate, and secures the permission of the called party to forward the voice message. The operator then commands the control system **102** to communicate the voice message stored in the outbasket to the called telephone line as indicated by arrow **114**.

For voice messages forwarded to another inbasket rather than to a telephone number, the voice message is not actually recorded in duplicate in each of the designated inbaskets. Instead, a notification is merely stored in the inbasket which indicates that a forwarded message is stored by the system for delivery to the owner of the forwarding message basket. The notification indicates the particular outbasket and the particular message within the outbasket which is being forwarded. This enables the person sending the message to retain ownership of the message in his own outbasket and selectively change or delete the message until it has actually been delivered. Depending upon the delivery instructions of the sender and the preselected instructions of the recipient, a forwarded message might simply wait for delivery until the recipient retrieves the messages stored in his inbasket at some point in time. Alternatively, the recipient might be informed of the

receipt of a message in his inbasket by a paging signal communicated over a paging system (not shown), by the illumination of an indicator light at the recipient's telephone, or by a telephone call to the recipient's telephone number informing the recipient by a prerecorded message that a message has been received in the recipient's inbasket.

The prompts and client greeting section of data store **104** stores a plurality of individually addressable voice message prompts explaining how to operate the voice service system **100** and a client greeting for each inbasket. A voice message prompt is prerecorded for each anticipated state at which a caller might access the voice service system **100**. These prompts provide an explanation as to how the user should proceed from the particular point of use and are accessed by the control system **102** and communicated to the user as appropriate. At any point, a knowledgeable user may override the prompt by inserting a command without taking the time to listen to a complete prompt message. The client greetings are provided as an answer mode for message storage accesses to each of the system inbaskets. Each client may record and change his own personal greeting at will. This enables the greeting to include current information such as telephone numbers at which the client can be reached for a given period of time, indications that the client is on vacation for a given period of time, indications as to when the client will return to his office and so forth. In the event that a client fails to have recorded a preestablished client greeting, a general system greeting is provided in its place. The system greeting invites the caller to leave a message but does not identify the specific owner of the inbasket which has been accessed by the call.

The audit trail portion of data store **104** stores a record for each caller accessing the system **100** of the command signals which have been given to the system **100** by the caller. This record enables the control system **102** to select particular voice message prompts in accordance with the current state of the calling line. In addition, in the event that a calling party requests operator assistance, the audit trail record is displayed on the video display unit of a selected operator console **102** so that the operator selected to give assistance can see immediately the state of the calling line, and what attempts have been made by the calling party to control the system **100**. This enables the operator to more readily determine what mistakes have been made by the calling party and what needs to be done to place the system in the state desired by the calling party.

The major functions which are performed by control system **102** are indicated by a plurality of functional blocks shown within the outline of control system **102**. The particular functions executed by control system **102** depend upon by which one of the functionally different types of telephone lines the control system **102** is accessed and upon which keyboard commands are entered by a person accessing the voice messaging system **100**.

A secretarial line is effectively an extension of a client's normal use telephone line. The client's line may be utilized for receiving and placing telephone calls in a normal manner. The control system **102** responds to an incoming call on a secretarial line by waiting for a predetermined number of rings which may be preselected by the client in accordance with the day of the week, and time of day, and then answering the telephone.

Upon answering the telephone as indicated by answering function **116** the client greeting is accessed in data store **104** and communicated to the caller. The caller is invited to leave a recorded message which is then recorded and stored in the client's inbasket if a message is generated. Because the caller could quite possibly be a person who is not a client of the voice message service system **100** and is totally unfamiliar with its operation, no prompts are provided to the caller with respect to the editing of any message which is left in the client's inbasket. Such prompts might prove to be bewildering and confusing to any nonclient caller. However, a sophisticated caller who is familiar with the voice service system **100** is free to use normal system editing commands which enable the caller to edit the voice message. Upon completion of the message or upon the occurrence of a client selected timeout duration, the call is terminated and the message remains in the client's inbasket until retrieved by the client.

Another type of line upon which a call might come into the voice answering system **100** is a direct incall line. A direct incall line is responded to by control system **102** with answering function **116** in a manner similar to a response to a call on a secretarial line. The principal difference between the secretarial line and direct incall line is that a direct incall line is dedicated to the particular inbasket of the client and is not available for general use by the client. Typically there is no reason for waiting for a specified number of rings before answering a direct incall line and such a line is answered as soon as it becomes active.

A general incall line is similar in nature to a direct incall line except that the general incall line is not associated with any particular message basket or inbasket thereof. Upon accessing the system **100** through a general incall line, a caller is prompted to enter a message basket number which number associates the incall line with a particular inbasket and causes control system **102** to transfer control of the call to answering function **116**. Operation then becomes functionally equivalent to the secretarial line and direct incall line except that at any time the caller may command a change function **118** which enables the caller to enter a new message basket number code and thereby associate the general incall line with a different message basket and enable the caller to leave a message with the newly selected inbasket in accordance with answering function **116**.

A general access line is intended primarily for clients of the voice service system **100** and affords the broadest range of system functions. Upon calling in on a general access line, a caller is prompted to either enter a message basket number if he desires to leave a message in another's inbasket or to enter his own personal ID number if he desires to have access to the ownership privileges of his own account as indicated by own account function **120**. If the caller elects the enter message basket number code function **120**, operation of the system is functionally equivalent to the response to a general incall line. Execution of a change function **124** enables the caller to select a new system inbasket in a manner functionally equivalent to change function **118**.

If the caller on a general access line selects the enter personal ID number code function **126** instead of the message basket number code function number **120**, the caller is granted immediate access to a message retrieval function **128** for the inbasket portion of his own message basket. The message retrieval function **128** informs the caller whether or not there are any messages within his

inbasket and, if there are, begins communicating the voice messages over the connected telephone line on a last in first out basis. Before each message is retrieved, the caller is informed of the age of the message on a lapsed time basis. For example, the system might inform the caller that the message was recorded fifteen minutes ago and then begin relaying the message. This lapsed time indication avoids any uncertainties which might arise from different time zones and the caller may obtain the exact time for receipt of a message with operator assistance. As a caller retrieves his messages, he may utilize the editing commands to rapidly scan through the messages before listening to the messages more carefully a second time or may on an individual basis command that each message be saved or cleared. After each message is relayed the caller may also direct, at the caller's option, a reply to the sender, a forwarding of the message to one or more other parties. The caller may also simply go on to the next message, with the present message being saved or cleared at the option of the caller. After reviewing the incoming messages, the caller is informed of the status of any outgoing messages in the caller's outbasket which are awaiting delivery. At any time during this process the caller may execute a change function **124** to leave a message at another's inbasket or an account activity change function **130** which enables the caller to select one of the ownership functions. The control system **102** retains the originally entered personal ID number code and does not require reentry of this code. Upon executing an account activity change function **130** the client may selectively return to the retrieval function **128** or may command a sending function **132**. In response to selection of the sending function **132** the control system **102** prompts the caller to record a voice message in the caller's outbasket by initiating a talk command. During the recording of such a message all of the edit functions are available to the caller. Upon completion of the message the caller enters a save command and is then prompted to enter an address code. The address code is a two digit code which identifies a preestablished list of up to 99 addressees for the voice message. Each of the 99 entries on the address list or distribution list may in turn be another list of up to 99 addresses, thus permitting a distribution list of almost 10,000 addresses in total. Upon selection of a distribution list, the caller is prompted to enter a one digit code selecting a preestablished set of delivery instructions for the message. For example, the instructions may specify that delivery be made only during certain designated days or times such as normal business hours in the calling party's time zone or a different time zone. Upon completion of the sending function the calling party may again execute a change command **124** or **130** to execute other voice messaging functions. For example, the caller may wish to access one or more of the administration functions which are available to an account owner. These functions include changing the greeting for the inbasket message, changing the conditions under which a secretarial line is answered and so forth.

The control system **102** also provides connection to a direct recall line which is the counterpart of a direct incall line in that it is associated with a particular message basket but enables the account ownership function instead of the answering functions. Upon answering a direct recall line, control system **102** executes the enter personal ID number code function **126** and functional execution, then proceeds in the same manner as if a caller on a general access line had elected the enter personal ID code function **126**. A general recall line is the counterpart to the general incall line and requires execution of an enter message basket code function **136** before advancing system control to the enter personal ID number code function **126** as with a direct recall line.

The direct recall line is of advantage in that it provides a higher security for access to the system because the caller must know both the telephone number of the direct recall line and the personal ID number code associated therewith. Each personal ID number code must be preceded by an asterisk symbol to identify it as such. Even further security may be provided by requiring a two field personal ID number code with the two fields being separated by a number sign (enter) key. The second field, if elected, may be changed at will by the client owner as one of the administration functions **134**.

The system architecture of the telephone voice service system **100** is shown in block diagram form in FIG. **2** to which reference is now made. The telephone network **108** provides a number of physically different types of telephone lines to which connections must be made by the service system **100**. By way of example, these different types are shown as including secretarial lines, direct inward dial lines, DX tie lines, and 2-wire lines. Connections are also provided for six wire operator stations. In the present example the different lines are shown to be connected rather arbitrarily to illustrate the maximum size of the system.

Up to 640 secretarial lines are connected to a concentrator **202** which selectively connects the voice information carried by these 640 secretary lines through 20 trunk lines **204** to a telephone room subsystem line group **1** processing circuit **206**. Only analog voice information is carried by the trunk lines **204**. Control commands and data such as trunk and line identification information, ringing signal indications, connection commands, and execution confirmation signals are communicated through a data set **208** within concentrator **202** over a serial data line **210** extending between data set **208** and a data set **212**.

A telephone room subsystem A, line group **2** processing circuit **214** is illustrated as connecting to a direct inward dial line while a telephone room subsystem D, line group **168** processing circuit **216** is shown as being coupled to one direct inward dial line, two DX tie lines, and two 2-wire lines. Each represented telephone line is assumed to be a bidirectional full duplex line.

The concentrator **202** and each of the telephone room subsystem line groups **206**, **214**, **216** are physically located at one or more telephone company central offices or client PABX centers. The system can accommodate up to four telephone room subsystems with up to 42 line groups being associated with each telephone room subsystem. Each line group can in turn accommodate up to 8 operator telephone lines and up to 22 telephone trunks or lines of another type. It is thus possible for each telephone room subsystem to connect to up to 1260 voice grade circuit terminations with the maximum of 4 telephone room subsystems providing in total connection to 5040 voice grade circuit terminations. Multiple lines may be assigned to a given data source to provide a capability of higher bandwidth than the single voice grade line. In the present example, the telephone room subsystem A line group **1** **206** is coupled through the 20 trunks **204** to concentrator **202** and also through a maxi-

mum of 8 operator lines **220, 222** to operator headsets **224, 226** at a plurality of operator consoles **106.**

The telephone room subsystems operate as interfaces between the digital portion of the telephone voice service system **100** and the analog telephone lines and trunks. They provide analog-to-digital conversion of the voice signals, detect and generate DTMF data and command signals, detect and generate dial pulses, and communicate the telephone line information to an associated real time subsystem over one of two redundant 2.048 megabit per second time division multiplex serial data channels. The 30 lines to which a line group processing circuit may connect are each assigned to a different voice channel while the control information for all of the lines as well as the line group is carried by a single channel. A 32nd channel is utilized to synchronize the serial data links.

In the present example telephone room subsystem **206** is coupled to real time system **1 230** by a pair of redundant serial data links **232, 233.** Similarly, line group **2 214** is coupled to real time subsystem **1 230** by a pair of redundant serial data links **234, 235** and line group **168 216** is coupled to real time subsystem **4 238** by a pair of redundant serial data links **239, 240.**

Up to 4 real time subsystems receive the voice and control data from the 4 telephone room subsystems, provide selected switching connections between channels, and communicate with an information processing system **250** for storage and retrieval of voice messages and system control. The real time subsystems also perform any signal processing such as silence compression upon the voice signals.

An interactive service subsystem **252** provides a communication connection between the information processing system **215** and input/output devices for the voice service system **100.** Interactive service subsystem **252** is illustrated as being coupled through two serial channel controllers **254, 255** and two sets of 16 serial data channels each **256, 257** to an RS **232** serial distribution panel **260.** Distribution panel **260** provides serial data connection to up to 32 different devices. It is representatively shown as connecting to a line printer **262,** a card reader **264,** to the keyboard display terminals **266, 268** and **270** within operator consoles **106** and to the data set **212.** It will be recalled that the data set **212** carries the control and data information between concentrator **202** and the information processing system **250.**

As shown in FIG. **3,** the telephone room subsystem A line group **1 206** includes a line group controller **302** connected between the 2.048 MBPS 30 channel synchronous data link **232** and an 8 bit parallel poled bus **304** and a second line group controller **306** connected between the 2.048 MBPS 30 channel synchronous data link **233** and an 8 bit parallel poled bus **308.** Only one of the line group controllers **302, 306** provides active communication with the real time subsystem **1 230** at any one time. In the event that the active line group controller fails, the other immediately assumes the duties thereof to continue uninterrupted communication between the connected telephone lines and the information processing system **250.** The active line group controller poles the line interface boards connected to the associated bus for voice channel data and upon receiving data, inserts the data into a preassigned one of the 30 time division multiplex voice channel data slots on the synchronous data link **232** or **233.**

Each line group provides bus **304, 308** connection to up to 15 line interface boards, each of which may couple to two different analog telephone lines. Each of the line interface boards is generally similar in construction except that certain variations are required in order to interface with the different kinds of telephone lines to which a line interface board may connect. A variety of different line interface board types have been illustrated in FIG. **3** to demonstrate the different types of boards which might be included in a voice for service system. For example, line interface board **1 310** is an operator type of line interface board and is somewhat simpler than other types of boards in that it need carry no signaling functions since these are accommodated through the keyboard display terminals such as terminal **266.**

The line interface board **310** interfaces two separate operator positions to the real time subsystem **230.** Line interface board **310** provides battery feed circuits for powering two operator headset microphones at each position and amplifier circuitry for driving two sets of headset earpieces at each position. Functional circuits located on line interface board **310** include redundant power supply inputs, two separate current limited −48 volt battery feeds per circuit, a circuit providing a side tone fixed at −26 db for each headset circuit, a circuit providing audio mixing of headset microphone inputs, a circuit providing 4-wire operation separate receive and transmit voice paths, and an onboard MU-LAW codec. The MU-LAW codec provides conversion between the 13 bit digital sample of an analog voice signal and an 8 bit byte representation thereof to enable a single byte of sampled data to have a greater effective range and resolution than would be possible with 255 equal magnitude increments. MU-LAW codecs are well known to those skilled in the art and are not further described herein. Each of the 6-wire interface connections of the line interface board **310** provides headset A transmit tip, headset A transmit ring, headset B transmit tip, headset B transmit ring, headset A and B receive tip and headset A and B receive ring. Two low impedance microphone inputs per circuit (600 ohms or less) will drive two 300 ohm low impedance headsets per circuit.

A concentrator line interface board **2 312** provides connection to two concentrator trunk circuits **204.** Concentrator line interface board **312** terminates to "dry" (no line voltage) 2-wire links from concentrator **202.** Because system control signals are communicated through the serial data link **210,** the line interface board **312** carries no supervisory or control signals except DTMF signals which are communicated through the telephone line. Line interface board **312** includes DTMF signal generating and detecting circuits which respond to or generate the required DTMF signals. These signals are separated from the voice channel information and are communicated to the appropriate line controller during a separate control time slot on the buses **304, 308** and are communicated by the active line group controller to the associated real time system during a separate control information time slot which occupies a 31st channel position on the synchronous data links. The line interface board **312** includes the redundant power supplies and MU-LAW codecs which are found on the operator line interface board **310.**

A direct inward dial line interface board **3 314** terminates two 2-wire DII trunks, handles all DTMF signals associated therewith, detects supervisory and control signals and incoming dialed digits. The two incoming lines may be either CO WINK-START or immediate

start lines. The direct inward dial line interface board **314** hosts DTMF detection and generation, dial pulse detection, tone and voice envelope detection, and audio switches for call progress tone insertion. Line interface board **314** further includes 48 volt battery feed circuits, redundant power supply input, secondary voltage hazard protection, MU-LAW codecs, 2-wire to 4-wire conversion, and 600 ohm or 900 ohm line impedance selected by a strapping option. Also included on each DID line interface board **314** is a fixed compromise network, a loop current indicator light emitting diode, a wink and immediate start strap option, a reverse battery indicator light emitting diode, a reverse battery front panel switch and test jacks for 2-wire testing. CO tip and CO ring wires are interfaced for 2-wire circuits. A loop start/ground start line interface board **316** terminates two pay station telephone number conventional lines. It can accept incoming calls as well as seize a line to dial outgoing calls and handle all DTMF, supervisory and control signals. A 2-wire line interface board **318** interfaces two CO loop-start or ground-start trunk lines. The 2-wire interfaces provide CO tip and CO ring. The onboard functions include DTMF detection, tone and voice envelope detection, 2-wire to 4-wire conversion, loop disconnect dialer, a loop-start or ground-start operation which is DIPSWITCH SELECTABLE, a 600 ohm or 900 ohm strap selectable line interface, a fixed compromise network, a loop current indicator light emitting diode, test jacks for 2-wire testing, redundant power supply inputs, secondary voltage hazard protection, and onboard MU-LAW codecs.

The real time subsystem **230** is representative of each of the real time subsystems and is shown in FIG. **4** as including a minimum of two and a maximum of 46 external transfer switches **402, 404**. Each of the external transfer switches is coupled to both an S bus which is controlled by a real time executive S **406** and a T bus which is controlled by a real time executive T **408**. At least two and not more than 22 real time processors **410, 412** are also coupled to the S and T buses. The real time processors **410, 412** are identical to the real time executives **406, 408** except that the bus control functions are not implemented on the real time processors **410, 412**. The S and T buses are each time division multiplex real time buses which are 16 data bits wide and operate in repetitive frames with 512 slots per frame. Only one of the buses is operative at any given time with the other being available as a hot standby in the event of a failure on the first bus.

The external transfer switches **402, 404** provide an interface between the 2.048 MBPS serial data links and the S and T buses **414, 416**. Each external transfer switch **402, 404** may connect to two high speed data links, each of which carries 30 bidirectional voice channels and one bidirectional control data channel on a time division multiplex basis. In response to system commands the external transfer switches can connect any incoming or outgoing voice channel to any one or more time slots on the S bus **414** or T bus **416**.

An any channel to any channel connection scheme thus becomes possible. For example, certain incoming voice channels can be connected to a bus time slot allocated for voice message recording or selected outgoing channels can be connected to a bus time slot allocated to voice message retrieval. An operator line simply appears as one of the voice channels so that an operator can be selectively included in a set of voice connections. A multiparty conference call can be established by simply creating a bus channel for each incoming line and then connecting each outgoing line to all of the corresponding bus time slots for the incoming lines of the other parties. It will be appreciated that a "connection" does not imply a continuous physical connection but only the transfer of voice sample data bytes between select serial data link time slots and select S or T bus time slots.

The real time processors **410, 412** are each comprised of a standard processor module with a real time extension board connected thereto to provide additional processing and data storage capacity. In the voice messaging environment the real time processors provide data compaction by converting PCM encoded bytes of data to run length in coding format and by detecting periods of silence and encoding such periods in a run length encoding format. Data corresponding to periods of silence in excess of one second may be discarded if desired.

As with the real time processors **410, 412**, each processor connected to the X and Y executive buses is comprised of a standard processor module and an extension module which extends the processor module and adapts it to a particular function to which the processor is to be dedicated.

A standard processor module **500** is illustrated in FIG. **5**, by way of example as including a port control unit **502**, a CPU **504**, a 4k×16 RAM program memory **506**, an 8K×16 resident executive ROM program memory **508**, and a 4K×16 scratch pad memory store **510** all interconnected by a 16 bit internal data bus **512**. The port control unit **502** provides connection to the X bus through a 16 word X inport stack **516** and an X outport stack **517**. Connection to the Y bus is through a 16 word Y inport stack **518** and a 16 word Y outport stack **519**. The inport stacks **516, 518** each buffer a 16 word data packet as a packet is transferred from the bus to the port control unit while the two outport stacks **517, 519** buffer a 16 word data packet as the packet is being transferred from the port control unit to one of the executive buses.

The CPU **504** provides a basic data processing capability and may have its instruction set extended for special data processing functions by an extension module which connects to the standard processor module **500** and dedicates the standard processor module **500** to a particular processor type.

The 4K scratch pad memory store **510** provides storage for a large number of system variables and permits different sections or pages thereof to be dedicated to particular processes or programs. This eliminates the need for much of the time consuming process of storing process variables for one process or program whenever it is interrupted by another process or program. A basic REX program memory **508** is implemented in ROM to provide on the standard processor module **500** basic executive service functions such as a bootstrap startup program functions, diagnostic analysis, and communication over the executive buses. The 4K×16 RAM program memory **506** permits the standard processor module **500** to receive and store additional program data from other sources such as disk files on an overlay basis.

The standard processor module **500** is a complete computer constructed on a single board. When augmented by one of the several extension unit types, it operates as a processor on the X and Y executive buses. The standard processor module **500** serves as a self-contained functional node in an array of such units inter-

linked via the two independent, very high speed bidirectional X and Y data buses. The X and Y buffered data ports operate asynchronously relative to the standard processor module **500** itself. With some exceptions, a full instruction cycle of CPU **504** is 133 nanoseconds, including all accesses to program memory, working registers, port input/output buffers stacks and scratch pad memory.

All standard processor modules connected to the main X and Y buses are logically isolated therefrom by the logic of the port control unit **502**. Interprocessor transfers are effected in packets of 16 16-bit words, moved between respective output and input stacks at the instantaneous rate of one word per instruction cycle. Including all overhead, each X or Y main bus in the standard processor module array can maintain an average data rate of about 40 million bits per second. The physical identity of a standard processor module is determined by a 7-bit code permanently wired into each connector (permitting a maximum of 128 boards of all types on the main bus). For communications between processors, a logical bus identification (BID) is used within packets and subsequently translated to the appropriate physical BID immediately prior to packet transfer.

The real time executive **408** for the T bus **416** shown in FIG. **4** is illustrated in greater detail in FIG. **6**. The real time executives **408** and **406** are essentially the same except that by a strap selection option one is designated to control the T bus **416** and the other the S bus **414**. Furthermore, except for the addition of a small amount of bus control circuitry such as synchronizing crystal clock signal generators, the real time executives **406**, **408** are the same as the real time processors **410**, **412**.

Referring now to FIG. **6**, the real time executive **408** includes a standard processor module **602**, a real time processor extension unit **604**, and an internal transfer switch **606**. The real time processor extension **604** connects to the standard processor module **602** by the 16-bit internal data bus **612** of standard processor module **602** and includes 24K words of additional program memory **614**, 64K words of additional data memory **616**, and a direct memory access (DMA) controller **618**. The data memory **616** includes a 1K word section **620** which is dedicated as a DMA control memory for the DMA controller **618**. This section **620** stores buffer descriptors for use of DMA controller **618** in executing data transfer operations.

The DMA controller **618** operates on a stand-alone basis to transfer data between selected channels of the S and T buses and selected system storage locations such as records within magnetic disk files. The DMA controller **618** is coupled to the internal data bus **612** and by DMA channels **620** to the internal transfer switch **606**. DMA controller **618** is also coupled to communicate CPU interrupts to the standard processor module **602** in order to selectively interrupt the CPU of the standard processor module **602** as necessary to obtain communication over the X and Y executive buses.

The internal transfer switch **606** includes a DMA channel interface **624**, port command memories **626** and **512** word×16 bit S and T bus buffer stores **628**, **630**. The internal transfer switch **606** is similar to the external transfer switches **402**, **404** (FIG. **4**) and operates to transfer data between selected frame slots on the X time division multiplexed S and T buses **414**, **416** and selected DMA channels **620**.

Referring now to FIG. **7**, the interactive services subsystem **252** includes an interactive services executive **702** coupled to control an 8 bit parallel poled bus designated U bus **704** and an interactive services executive **706** coupled to control an 8 bit parallel poled bus designated V bus **708**. The two interactive services executives **702**, **706** and their respective buses **704**, **708** provide redundant coupling of data information between the X and Y executive buses and the various input/output devices of the interactive services subsystem **252**. By way of example, these devices are shown to include the serial channel controllers **254**, **255** which interface the U and V buses to the 16 channels **256** and the 16 channels **257** respectively. As shown in FIG. **2**, these channels connect in turn to the RS **232** distribution channel **260**.

The U and V buses are also shown as providing a redundant coupling to a magnetic tape controller **714** which in turn couples through a magnetic tape formatter **716** to two magnetic tape drives **718**, **720**. The I/O devices connected to the interactive services subsystem **252** have been illustrated by way of example and particular I/O devices can be added to the subsystem or deleted in accordance with the objectives and requirements of a particular configuration of a telephone voice service system **100** in accordance with the invention.

The interactive services executive **702** includes a standard processor module **724** and an interactive services subsystem bus extension unit **726**.

Referring now to FIG. **8**, the interactive services subsystem processor extension unit **726** includes a microprogrammed interactive bus controller **802** which couples to the U bus **704** and V bus **708** and is selectively operable to control the U bus **704**. The interactive bus controller **802** couples to an interrupt FIFO store **804** which in turn couples interrupt requests to the CPU of standard processor module **724** to control the transfer of data between a 64K×16 data memory **806** and the executive X, Y buses. The data memory **806** is partitioned to include a 4K poling list which is coupled for communication of buffer and controller descriptors with the interactive bus controller **802** and a 60K data memory section which is coupled for communication of I/O data with the interactive bus controller **802**. Data memory **806** is further coupled along wth a 20K×16 program memory **808** to the internal data bus **812** of the standard processor module **724** to which the extension unit **726** connects. The program memory **808** includes a 12K×16 program memory store **810** composed of random access memory for receiving overlay programs and an 8K×16 extended resident executive ROM **812** storing process programs which are specifically related to the interactive services subsystem.

Referring now to FIG. **9**, the information processing system **250** includes in addition to the interactive services subsystem **252**, the real time subsystems **1–4 902**, an executive services subsystem **904**, an information processing subsystem **906**, and a filc services subsystem **908**. The executive services subsystem **904** includes the two 16-bit parallel executive services main X and Y buses **912**, **914** respectively, an executive services processor **916** coupled to control communications over the X bus **912** and an executive services processor **918** coupled to control communication over the Y bus **914**. The executive services processor **916**, which is substantially identical to processor **918** except for connection to control X bus **912** instead of Y bus **914**, includes a standard processor module **920** and a bus controller exten-

sion **922**. The dual X and Y bus arrangement provides redundancy in the event of a failure associated with one of the two buses **912, 914**. However, the two buses are operated independently of each other and carry separate, not redundant data. However, in the event of a failure of one of the buses **912, 914**, the other connects to all of the subsystems of the information processing system **250** and can carry the data associated therewith.

The executive services subsystem **904** manages the interprocessor communications and that part of the system software that is responsible for systemwide resource and activity management.

Referring now to FIG. **10**, the bus controller extension **922** includes a W port **1002** which connects to the particular X or Y bus which is being controlled. W port **1002** is controlled by a microprogrammed packet switcher **1004** which poles the data output ports connected to the controlled executive bus and upon finding a port with a 16-bit packet stored therein awaiting delivery, receives the logical address stored in the first 7 bits thereof and uses the logical address to access a 4K×12 control memory **1006** which stores a table converting the logical address to a physical address corresponding thereo. The microprogram packet switcher **1004** then tests the indicated physical address input port for availability to receive the packet of data. If available, the packet is transferred. If not available, the packet to be transferred is queued in a queue of processors attempting to transmit to the busy recipient. If the processor will not accept any packets, the job services and management program (JSAM) determines whether a failure has occurred and if so invalidates the symbolic and logical identities of the processors stored in control memory **1006**. This has the effect of logically removing that processor from the system.

The bus controller extension **922** also extends the storage capacity of the executive services processor **916** by the addition of a 16K×16 program memory **1008** and a 16K×16 data memory **1010**. A 16 bit internal data bus **1012** interconnects the components of the bus controller extension **922** and to each other and with the standard processor module **920**.

Referring now to FIG. **9**, the file services subsystem **908** provides the primary high capacity permanent storage media in the form of one or more disk drives **930, 932**. The file services subsystem **902** includes at least two disk data processors **934, 936** which interface the disk drives **930, 932** to the X and Y executive buses **912, 914**. The disk data processors **934, 936** receive data access requests for transfers of data over the executive buses and satisfy those requests by transfers of data packets between the disk drives **930, 932** and requesting processes over the executive service buses **912, 914**. Each of the disk data processors **934, 936** includes a standard processor module **938** and a disk controller extension unit **940**.

Although not separately shown, the disk controller extension unit **940** provides a 4K×16 PROM for program memory for physical initialization of the disk data processor **934**, wakeup processing, and the disk controller microprogram. An additional 12K×16 words of RAM program memory are made available to augment the RAM program memory on the standard processor module **938**. The RAM program memory of the disk data processor **934** is used for transient programs (infrequently used disk logical I/O routines), file management programs, and special purpose disk data processing programs such as the message basket maintenance programs, the SYSDISK programs, the system file maintenance programs. Any unused RAM program memory is available for general purpose programs that can run in any delta processor with sufficient program memory, scratch pad, and data memory resources.

The disk controller extension unit **940** further includes 64K×16 words of data memory, all of which is acceptable by programs executing in the disk data processor **934**. The disk controller extension unit **940** and the data memory therein provides a direct memory access DMA interface for the disk drive control logic. The disk data processor **934** provides standard data memory management routines and deals with blocks of data memories of up to 4K words in length. Disk data blocks longer than 4K words are supported by the data-chaining features of the DMA interface and of the REX I/O service routines of the disk data processor **934**. A buffer controlled block within the data memory lists the areas allocated to buffer the disk data blocks. Disk data buffers are maintained as a transparent "cache" memory for the data accessible on the disk drives attached to the disk data processor **934**. Disk data process buffer management routines return unused buffer space to the data memory master only when the data in the buffer is no longer valid or when available data memory is not sufficient for current demand. Unused buffers are maintained in a queue, the first entry of which is the least recently used buffer for cache management purposes.

A physical record is a minimum unit of data accessed by the disk data processor **936**. A physical record contains either two or three fields for identification and information storage, namely a count field, a key field (optional), and a data field. A gap between fields allows the disk data processor **934** to operate on the key or data fields after verifying the identity of the physical record. The data available for processing by the disk data processor **934** programs other than the disk controller microprogram is called a block, and is contained either in the data field alone or in the combined key and data fields. A block may be in one physical record or may be written across track boundaries in two or more physical records. A third unit of data is the logical record, which may be either part or all of a block or may be a series of blocks chained together by pointers. Programs outside the disk data processor **934** can act as only logical records, but all such accesses are translated into references to blocks within a specified area of the recording medium. On any initialized storage medium, the label record contains access information for a pack directory. The contents of the pack directory are records that describe the unused area of the pack (available tracks) and the separately allocated areas, which are referred to as data sets. The record in the pack directory that describes a data set is called the "data set label". It defines the characteristics of the data set and gives the location of two separate areas. The main area is the data area and contains the data blocks. An optional control area contains access and resource information that is automatically maintained by the disk data processor **934** programs and is only indirectly available during logical process of the data set. Each area is described in terms of the physical and logical organization of information in it and its location on the storage medium. Locations are defined in terms of extents, each of which is a continuous set of tracks. Access requests are always by block or track number relative to the start of the data set. Disk data processor programs translate the relative requests to the proper track and the proper extent.

The information processing system **906** includes at least two general purpose processors **942, 944** which provide a general multiprocessor data processing capability. For example, system accounting and administrative processing tasks would be assigned to the general purpose processors **942, 944**. The general purpose processor **942** is exemplary of these processors and includes a standard processor module **946** with a general purpose extension **948** coupled thereto. The general purpose processors **942, 944** provide program and data memory for executing system utilities that may be executed in any processor with sufficient memory resources. In addition, each general purpose processor defines one or two virtual machine types for execution of high level source language programs under a virtual machine interpreter (VMI). A kernel of standard processor module programs supplements the virtual machine interpreter to provide proper interfaces between the virtual machines and the actual information processing system **250**

The general purpose extension unit **948** provides additional program memory and a standard as well as an extended data memory interface. The general purpose extension unit **948** executes the standard data memory access instructions for the first 64K words of attached data memory. Data memory access registers and error detection and correction logic are provided as for other extension boards. Data memory itself is on separate data memory extension units **950**. Access to the whole of the attached data memory is provided by extended data memory extension instructions. These allow 22 bit word addressing via the standard processor module address registers. The general purpose pocessor **942** provides standard data memory resource management routines for the first 64K words and special routines for the rest of the data memory. Each general purpose processor may include up to 8 megabytes of data memory.

SYSTEM ARCHITECTURE

A. System Summary

The system **100** is a general purpose, multi-media computer system that uses fail-safe architecture to provide very high levels of availability and uninterrupted processing. Continuous operation for extended periods is assured, with no down time normally required for failures, maintenance or system modifications. The system **100** is a tightly-coupled, distributed network of multiple high speed processors, interconnected by a high speed packet switching network, and a fully distributed fault tolerant operating system that together provide a flexible processing system. The system **100** can be used in environments which mix real-time, computational communications, interactive and transaction processing with large numbers of peripheral devices and storage units. The system **100** provides for a flexible growth path which is independent of the initial system configuration.

B. System Architecture

A Delta system consists of five functionally unique subsystems; executive services, information processing, file services, interactive services and real-time. Each subsystem contains at least two identical processors, with the capability to expand individual subsystems as required up to a maximum of 32 processors per system **100**. This unique architecture provides maximum flexibility in supporting multiple concurrent applications.

There are three different perspectives in viewing the architecture of the Delta System. The physical system has a hierarchical structure composed of subsystems, plus their devices, each incorporating specialized processors, functional characteristics and organizations. The functional organization includes a hierarchical network of system processes that provide an open-ended environment for large numbers of concurrent and simultaneous application processes.

The user system designer's structure provides a network consisting of a multi-processor host system that executes the machine code of the hardware processors and a number of idealized virtual machines that execute a higher level source-language oriented instruction set.

The following sections describe these aspects of the system.

C. The Physical System

The system **100** consists of from eight to 32 processors, together with a range of controllers, peripherals and storage modules. Each processor is a fully independent, high-speed 16-bit machine having a non-microcoded architecture and high-speed program memory. A maximum of 7.5 million instructions per second can be executed with an instruction cycle time of 133 nanoseconds. The instruction set contains over 340 instructions.

Each of the five subsystems within the system **100** consists of customized processors suited especially for the functions of that subsystem. Each processor type consists of a common processor to which are added the following extensions: extra memory, microprocessor based device controllers, interfaces to other system bus structures, and extensions to the basic processor architecture. Although the type of processors varies by function, basic elements of the architecture are common to all processors.

The management services needed to run a processor are common to all processors and are hard-wired into the processor. Each processor also has a firmware-resident executive system (REX) which organizes and manages the resources of the processor on behalf of both the system and the active processes within the system. The resident executive provides a wide range of functions ranging from wake-up diagnostics, interrupt handling, timer management, to input/output services. In addition, each processor type has a customized extension to the executive, which manages the individual nature of the various processor types.

To be able to communicate with other processors in the system **100**, each processor has a pair of input/output ports that interface it to the packet-switched main interprocessor bus structure. Both buses **912, 914** of the pair are active, providing dynamic load sharing, thereby increasing system **100** utilization and throughput. Each provides a peak transfer rate of 120 Mbps and a sustainable rate of 40 Mbps.

To eliminate contention problems that can exist in multiprocessor systems based on global memory resources, each processor contains its own memory. Processors within the Information Processing Subsystem **906** can have from 500K-Bytes up to 8 M-Bytes of memory. All other proecessors may contain 500K-Bytes of memory. Independent processor upgrades can take place unrestricted by arbitrary system considerations. The total amount of memory available in a Delta System can be extremely large.

A similar philosophy exists within the processor architecture. Rather than have a small set of machine registers shared between the various activities and events contending for the use of the processor, with wasteful saving and restoring operations between every change, each processor is provided with over 4000

scratchpad registers to be allocated among multiple resident tasks.

Each of the five subsystems has sufficient resources to ensure survival of any single-point failure within itself, as well as many multiple-point failures. The system **100** as a whole is able to survive such occurrences.

The executive services subsystem **904** (ESS) manages the inter-process communication (IPC) network and that part of the system software responsivle for system-wide resource and activity management. Executivc services subsystem **904** consists of two executive serviccs processors **916, 918**. Each executive services processors **916, 918** controls one of the two interprocessor buscs **912, 914**. A bus controller extension **922** in each executive service processor **916, 918** includes a high-speed microprogrammed switch controller **1004** that transfers packets directly between the ports of the various attached processors. This intelligent packet-switcher **1004** translates logical packet addresses to physical destinations in the system, reports transfer failures to higher levels in the system, and optimizes traffic flow between system components. In addition to the two executive services processors **916, 918**, executive services subsystem **904** consists of the two interprocessor buses, the dual double-buffered ports in each processor, and the packet transfer service provided by the resident executive in each processor.

The executive services subsystem **904** bus controllers **916, 918** use adaptive high-speed polling techniques to achieve high sustainable data transfer rates. This mechanization enables the system **100** processes to communicate via logical physical-location independent addresses.

The information processing subsystem **906** (IPS) is the physical host to the ideal machines used for most application software within the system **100**. It is physically composed of at least two, and as many as 26, general purpose processors **942, 944**.

The information processing subsystem **906** provides the hardware and software capabilities needed by the application software to customize the system **100** to the user's requirements. The information processing subsystem **906** supports execution of programs written in Pascal.

Each general purpose processor **942, 944** in the information processing subsystem **906** can have up to 8 Megabytes of memory and host up to 255 ideal machines, each of which is allocated real memory and other resources as required by an application program. If a processor fails, programs can be rescheduled automatically in alternate hardware resources. This service, provided by the executive services subsystem **904** applies to all other software processes running in other system **100** processors.

The file services subsystem **908** (FSS) is responsible for the management of the disk storage mcdia and is composed of at least two and up to 26 disk data processors **934, 936** (FSPs) and their associated disk drives **930, 932**. At least two of the storage volumes attached to the file services subsystem **908** contain the system database, which is automatically maintained in duplicate. (This security service is also available to any disk or magnetic tape file.) The disk data processors **934, 936** contain microprogrammed disk drive controllers and a high-speed data channel **952** to manage traffic to and from the disks. Software in the disk data processors **934, 936** includes the physical input and output disk handlers, as well as the logical input and output processes that interface with the rest of a system **100**.

The disk data processors **934, 936** support any Control Data Corporation SMD or equivalent storage module. Certain modules permit dual-port attachment to two controllers.

The interactive services subsystem **252** (ISS) manages the transfer of data between processes and various peripheral devices that may be attached to a Delta System. The two dedicated processors of an interactive services subsystem **252**, the interactive services executives **702, 706** (ISXs), each manage a separate interactive bus **704, 708**. Attachable devices include magnetic tape drives **718, 720**, communication data channels, terminals, and printers. Many devices can be attached to the interactive services subsystem **252**. For example, up to 992 data terminals can be configured. Furthermore, multiple interactive services subsystems can be configured in a system **100**, each using its own processor paid and interactive bus pair, allowing nearly 4000 devices to be configured.

As mentioned, each interactive services subsystem **252** controls a separate interactive services bus pair, to which is attached dual-ported fully-buffered controllers for the various peripheral devices. Due to the sporadic nature of the data transfers between interactive devices, the interactive services executives **902, 906** use an adaptive polling technique and a multiplexed DMA channel controlled by a separate, independent microprocessor in the interactive services executive extension **726**. The microprocessor transfers data between memory buffers and devices at a rate determined by the individual demands of the various attached devices.

The real time subsystem **230** (RTS) provides the user with the unique capability to both switch and process real-time continuous data streams. Each real time subsystem is comprised of a pair of dedicated processors, the real time executives **406, 408** (RTX); a pair of synchronous time division multiplexed (TDM) parallel buses **414, 416** (each controlled and managed by a real time executive); a set of external transfer switches **402, 404** (ETSs) which map pulse-coded modulation (PCM) audio and high-speed synchronous data channels to the buses; and two or more real time processors **410, 412** (RTPs) which provide the capability to process data and effect data transfer between the external channels and other processes within the system **100**.

Up to 1260 channels can be connected to each real time subsystem **230** of which 480 can be active simultaneously. Data on such channels is normally PCM voice or similar, and bandwidth through the switch can be dynamically allocated to channels with higher requirements, allowing up to 60 MHz if necessary. Each external transfer switch **402, 404** permits the summing of output channels, permitting conferencing of multiple voice channels.

Real time information can be acquired or generated by the system **100** through the real time processors **410, 412** to perform tasks such as signal compression/decompression.

Up to four cross-connectcd real time subsystems **230, 238** can be configured in a system **100** to match applications requiring over 5000 real time attachments.

D. System Functional Organization

Functionally, the different processor types, memory, and peripherals may be viewed as a "pool" of manageable resources to which can be allocated various software processes that together support an application.

The software processors are grouped into five levels in a hierarchical structure where each level supports the higher levels. The five levels are defined as follows: Level **0**, primitive functions provide basic machine functions and interprocess communications. Level **1**, utility processes, provides file services, device handlers, and transient functions. Level **2**, system functions, provides operating system functions such as job scheduling, file management, etc. Level **3**, user applications, represents application programs developed by users. Level **4**, subordinate processes, provides subroutines and subordinate processes used by application jobs.

Levels **0**, **1** and **2** collectively are referred to as the kernel system. Levels **3** and **4** are associated with application software. Within the kernel, and responsible for managing each processor, is the resident executive (REX). Each processor in the system has its own REX. Also provided at this functional level is a method of communication between logically identified software processes which may reside anywhere in the system. Communications between all processes is through discrete packets on the high-speed packet switching network. Each pack or set of packets is logically addressed to a destination process. A transmitting process may request a response/acknowledge packet to be returned.

These process levels are ach described in greater detail in the following paragraphs.

Level **0**, the primitive function level of the system **100** is composed of the following: (1) basic hardware capabilities of the system **100** processors, including the controllers, interfaces and memory extensions provided by the various extensions types; (2) functional enhancements and management services provided by REX in each system **100** processor; and (3) the interprocess communications services accessed via REX in each processor, and managed system-wide by the executive services subsystem **904**.

Level **0** supports the set of software processes that constitute both the kernel system software and whatever application systems are running in the system **100**. To the system designer, the logical environment provided by Level **0** is in many ways similar to that provided by the system executive of a more traditional single-processor machine, in that it provides the ability to run a set of inter-communicating processes that can be built into various operational layers.

Level **1** processes include such functions as device handlers, file services, specific device controllers, and transient processes invoked and controlled in the same manner as device controllers. They provide logical interfaces and handlers to specific hardware attachments, and therefore their location is fixed in the system with access controlled by higher-level system software. If a new device type is attached to the system a process at this level must be written to logically interface the new device into the system. Communications access to Level **1** is granted only by Level **2** processes; once granted, packet transfers are direct until the communications link is terminated.

The major processes of Level **2** system functions form the operating system of the system **100** to which applications and other system processes make requests for services such as ownership of devices, opening and closing of channels, and file management. The logical addresscs of these processes are known globally to all users so that Level **2** functions provide the resource and activity management on a system-wide basis, together with access control for Level **1** functions. Level **2** functions include the following: job scheduling and allocation, device management, system directory and volume management, system event processing, and system management.

An application from Level **3**, applications job supervisors, may consist of a single process or a complete network of concurrent processes. Each job and job supervisor is created as a result of an external or timer-generated stimulus, and is terminated when all processing requirements have been met or prove impossible to meet. Whether simple or complex, each invocation of a job is given a unique identifier called a job number. For each job number there is typically a unique Level **3** process invoked, termed the "primary process". The Level **3** job supervisor is the mechanism which tailors the system **100** to a particular job and is the main environment in which the systems designer work. The Level **3** job could invoke the existence of (Level **1**) transient processes during its lifetime to provide special services. A job could also invoke another Level **3** process, for example where complex checkpointing and fast response to failure are required.

In some instances it is possible for a Level **3** applications process to act as a host for subordinate processes. For example, if the Level **3** task is the program development system (PDS), the invocation of the editor or compiler, which runs under the program development system, is an example of a Level **4** process. The Level **4** process communicates directly with its supervisor without recourse to the interprocess communication network as it exists in the machine space allocated to the supervisor process.

At a given time, a Delta System may be supporting a large number of application jobs resident in multiple application processors and/or co-resident within the same applications processor. In this multi-process, simultaneous processing environment, each job is a systemwide network of level **3** and **4** processes, identified by a unique job number, making use of level **2** and level **1** system services as required. In a command and control application, for example, initiation of a single job process network could link a terminal with a voice line and a set of files to provide a single operator with a dedicated command console. If the system supported 20 such consoles, 20 similar concurrent job process networks could exist. (Note that common program code would be shared by the job processes wherever possible for efficient use of memory resources.)

Another example will help to illustrate some system components involved in the management and support of applications. Mixed with the main tasks of the system **100** can be terminals configured as program development stations for ongoing system development. An operator on one of these terminals can invoke a program development system to be attached to the terminal, providing interactive editing and program source file compilation, plus a full range of utility functions.

A sequence of events that might occur during the operator's interface with a program development system terminal is as follows:

1. The "RESET" character from the unassigned terminal is sent to the terminal handler (Level **1**) as a message.

2. The terminal handler, recognizing that the terminal is currently "un-owned" by any job, sends message to the terminal prompting the operator for the initial log-on sequence.

3. Upon receipt, the terminal handler sends the identity of the terminal and the operator-entered log-on sequence to the system device log-on process (Level 2).

4. The system device log-on process consults its file and finds that this terminal permits log-on to the requested application system by a qualified user.

5. System log-on sends a "Create Job Request" to the system job scheduler (Level 2), including the terminal identity and the entered identifier of the desired application system, in this case program development system.

6. The job scheduler locates the program development system (Level 3) descriptor in the system program library and finds within the system **100** a location that provides the appropriate resources for the program development system process to execute (if possible, using an existing copy of the program development system program code). An idealized machine is created, allocating unique memory to the job and access to the program code. Then the program development system job is initiated with its own unique job identity. The terminal ID is passed to the program development system job as part of its start-up parameters.

7. The program development system then requests ownership of the terminal from the system device manager (Level 2) which is responsible for management of all attached devices (including read-time channels).

8. The system device manager records the owner of the terminal in its tables, and informs the terminal handler of the job number of the program development system that now owns the terminal.

9. A logical channel is thus established between the program development system and the terminal via the terminal handler (see FIG. 9). The system device manager is no longer involved until the terminal is terminated and released by the program development system.

10. Via interactive commands from the user, the program development system can initiate utility programs to run within its allocated resources, for example screen editors, compilers, assemblers, and linkers.

Two important principles should be noted from these examples, (1) The physical location of a process within the system **100** multiprocessor environment is not critical to the operation of the system and (a) all communications between system and user processes is by means of packet exchanges, even if processes happen to be co-resident in the same processor. Most of the Level **0** system functions and services are performed by a resident executive (REX) within the standard processor module of each system processor. A ROM copy exists in each processor to provide basic services to effectively manage the processor within which it is resident. The services include: interrupt handling, event management, timer management, memory management, process management, status monitoring, I/O service functions, list processing, inter-process communications, traps, wake-up and diagnostics.

These services together with the hardware provide the primitive functions available to the privileged processes in each processor. Note that REX provides access to the executive services subsystem **904** which mechanizes the dual inter-processor buses **912, 914** that connect the multiple processes of the system **100**. There is also a unique extended resident executive (EXREX) for each type of processor. EXREX extends the services functions of REX to include features peculiar to each of the processor types. These include: physical initialization, process type model and version indications, management of extended memory, mechanization of processor peculiar functions, and interface with any auxiliary microprocessor.

At Level **1** the device handlers effect the logical and physical I/O and control of the various devices that can be attached to the system **100**, including: disk drives, magnetic tape units, line printers, smart terminals, dumb terminals, voice channels, operator stations, remote line concentrators, and discrete signal controllers. Device handler processes are intimately connected with the hardware they control and reside in the subsystem to which the devices are attached. Initial contact with a device handler by an application is via a Level **2** system process.

The Level **1** system functions also include utilities and transients which control various system and user processes such as voice compression and record; voice decompression and playback; spooler, etc. These processes may reside within either the host space or ideal space of the system **100**. Transient processes are created on demand and are dedicated to a particular job. Resident processes are shared. Creation, if appropriate, and access rights, are effected via Level **2** processes.

Among the Level **2** system functions job scheduling, allocation and monitor (JSAM) is a major system process that manages the allocation of processes to available processor resources. JSAM is responsible for creating jobs, monitoring the status of resources, recovering jobs when resource failure occurs, and removing jobs from the system when they terminate. System device manager (SYSDEV) receives requests to ownership of any attached device by a job. SYSDEV manages the allocation and overall configuration of the system devices. Once the allocation has been made, the job owning the device or channel communicates directly with the level device handler. System directory manager (SYSDIR) performs the management of all permanent storage resources including the maintenance of user data sets on both magnetic tape and disk, either on mounted volumes or off-line. Once linkage to a data set is made by a job, communications are between the job and the Level **1** logical I/O handler of the particular file service processor responsible for the volume. Access and control protocols for both datasets and devices are common at the user level (Level **3**). System device log-on manager (SYSDLO) receives any activity on devices not owned by a particular job from the handler for that device. By consulting its tables, SYSDLO can decide what action should be initiated when a given combination of external, unsolicited stimuli occur. This may result in a request to JSAM to start up a job in response to the event. Various start-up parameters are passed to the job to link it to the external events which it has been created to service.

System monitor (SYSMON) provides the normal system management functions, including general system monitoring, reporting, and housekeeping. SYSMON has many privileges available to it that are not available to Level **3** supervisors. As such, a stricter log-on procedure is used to gain access to this system. SYSMON maintains a dynamic short-term database of recent system activities. SYSMON permits activation of Level **3** supervisors to provide qualified terminals with access to all or part of the dynamic database. Such Level **3** programs conform to the requirements of the application system being supervised. Still another Level **2** system function, system event logger (SYSLOG), maintains a database of all logged events in the system **100**. This log can be used to generate application and system related

information. SYSLOG dates and time stamps each event and stores the event record on a disk dataset.

Level 3 provides the system application programs. A developed application, when scheduled by JSAM, is given a unique job identification during the lifetime of its invocation. A large number of identical jobs can coexist to service multiple users and a variety of terminal devices. A job can consist of single or multiple processes (all bearing the same job ID) and a job can dynamically create and destroy processes during its lifetime (typically, transients at Level 1). The individual processes of which a job is composed (minimum of one) can be distributed across the system **100**. The program development system (PDS) is an example of a Level 3 program and provides a complete interactive program development system that provides a single-user environment for the development and testing of application programs. Multiple program development systems can be run concurrently on the system **100**, alongside the main application jobs of the system. Included in the program development system are several Level 4 processes which support the program. These include a screen editor, a Pascal compiler, an SPM assembler, a linker, a debugger, and various utilities.

E. System Program Architecture

Some Level 1 and 2 processes can exist in any subsystem processor. When any process is to be initiated by the kernel system, its resource requirements are first analyzed, and the system initiates the process in whatever resource is available and appropriate.

Level 3 job supervisors, however, represent the major processes of applications. As such they normally reside only in the general purpose processors **942, 944** of the information processing subsystem **906**. A consideration of the program architecture of the system **100** explains why this occurs.

Processors comprising the system **100** are designed to be task specific. The system **100** processors have a common instruction set as well as a type dependent superset. The individual functions may include access to a special set of control registers, a small set of extra instructions related to a direct memory access (DMA) channel, or a set of special functions. In brief, the programming environment within all processors controls a main core of identical capabilities with subsets of special capabilities unique to each processor type.

The instruction set of the system **100** processors optimize execution speed by using a 16-bit instruction. This instruction is both a macro-instruction and a micro-instruction, in that a machine cycle and an instruction cycle are nearly always equal. A very close relationship exists between each bit in the instruction and the actions within the processor hardware, although there is a layer of translation. This results in assembler generated code producing very fast programs, which are ideal for processes responsible for real-time control and management functions such as exist in the real-time subsystems **902**, interactive services subsystems **252** and file services subsystem **908**. The system **100** instruction sets provide hardware control for both processors and various attached devices.

Wherever possible, Level 3 processes in the system **100** are written in a high-level systems programming language both for ease in the implementation and maintenance of complex applications, and excellent runtime integrity. However, to compile high-level programs into low-level instructions codes which will run alongside Level 0, 1 and 2 processes in an unprotected machine environment requires careful implementation by a skilled user.

The system **100** provides the user with a fault tolerant environment in which to run applications. In a typical system, program development occurs along side the use of the system for its prime and, presumably, high-availability tasks. New applications programs often incur run-time errors. Active jobs have to be both protected from having their code or data destroyed by other concurrent jobs, and prevented from inadvertently modifying other processes. Application program protection is assured with the system **100**.

Two features which satisfy the above requirements in the general purpose processors **942, 944** are:

1. A large memory space of up to 8 Megabytes per general purpose processor is provided for loading program and data segments of processes, togehter with a mapping system that allows the memory space to be assigned to processes in protected, controllable spaces. This real memory space has a contiguous address range, as viewed from the process to which it is assigned, but, through the mapping scheme, physically comprises demand-allocated, noncontiguous blocks of 4096 bytes.

2. Enhancements to the basic log of the standard processor module (SPM) architecture allow normal high-speed program memory (50 nsec access time) to be used as a writeable control store (WCS), with the basic standard processor module order codes executing like microinstructions. This function is supported by the addition of a hardware instruction decode table, additional instructions in the standard processor module instruction set, and a number of extra machine registers.

These two enhancements have a beneficial effect. Non-native instructions set can be executed in the general purpose processors **942, 944** by loading micro-code emulation sequences into the writable control store. An example of a non-native instruction set is a transportable P-Code set which may be produced by a higher level language such as Pascal or FORTRAN. The general purpose processors **942, 944** and their nature instruction set act as a host for the "ideal" machine which implements the higher level language through an "ideal" machine instruction set. Due to the very high speed of the host machine, the "ideal" machine instruction cycle time is generally faster than similar 16-bit micro-coded CPUs. In addition, the writable control store retains flexibility for future incorporation of other non-native (ideal) instruction sets, optimized to different task environments. As a system **100** can incorporate a number of general purpose processors (up to 26), it is conceivable that different general purpose processors could be dedicated to different applications each with a different "tailored" high-level instruction set.

Programs encoded in the ideal machine instruction set reside in the extended memory of a general purpose processor. Each process assigned to a general purpose processor is initially given a scparate memory space to operate in. The ideal machine instruction set of which the program is comprised will not allow addressing outside the range of the process' allocated ideal address space. Thus the process is both protected from programs in other ideal spaces and prohibited from modifying any other process' space. It can use only the instruction set of the ideal machine, which, for non-privileged uses, contains no method of accessing either the basic support hardware, or other processrs in the system **100** except by restricted access to a limited set of the primitive functions at Level 0.

The ideal machine instruction set and the ideal memory space collectively form an ideal machine (IM). An ideal machine provides a single user a uni-processor environment for programs written in high-level languages, with logical separation and protection for multiple concurrent processes running in other ideal machines co-resident in the same general purpose processor. From the programmer's view, the application task is written to run in a stand-alone machine with a formalized procedural interface to communicate with and use system **100** resources resident in separate, distinct machine spaces whether in other ideal machines or other system **100** processors. This meets the need to provide an efficient, protected environment for application processes.

The ideal machine instruction set is a pseudomachine code (or P-Code) derived from the P-Code originally defined by Wirth (Institute for Informatik, Zurich). The P-Code set is an idealized instruction set for stack-based machines, which are in turn the ideal target machines for block-structured languages such as Algol, Pascal, ADA, etc. These language types are the most efficient and productive languages for implementing complex systems.

Extended Pascal is the language supported on the system **100**. This is a superset of the ISO Pascal language with an extension that interfaces to the system **100** kernel software. These extensions also enable multiple Pascal processes to communicate with each other.

A single general purpose processor **942, 944** can support a large number of ideal machines, each one created dynamically to implement a Level **3** job. Thus if the system **100** is being used in a multi-terminal, single-function application, each terminal would be owned by a single-user job residing in its own ideal machine, with each job responsible for one terminal. As program code sharing occurs, and as jobs can dynamically acquire more data space, the actual number of ideal machines that can be accommodated per general purpose processor is application and time dependent, but 30 and 60 ideal machines, respectively, are typically low and high figures for one general purpose processor. Thus a system comprised of one hundred terminals using just three general purpose processors (to provide redundancy) is perfectly feasible.

The relationships between the program, the system **100** architecture and the system functions are such that:

1. The host program spaces of the standard processor modules (File Services, Interactive Services, Real-Time) are considered privileged, and designing processes to run in these machines requires a careful consideration of the standard processor module and REX architectures to ensure that system processes co-resident in this space are not affected.

2. It is feasible to write a major system task to run in the host space at Level **3** in critical real-time jobs, as the system software makes no specific distinction between jobs and their relative hierarchical responsibility as outlined in the preceding descriptions. The system **100** requires only those parameters related to the resources the various job processes need (that is, an IM or a host machine, and if a host machine, what type, if type-critical).

3. All of a general purpose processor's host programming space is taken up by the ideal machine monitor (IMM), which includes the P-code emulation microcode sequences, extension memory resource management, IM process management, and interface to REX basic services. The IMM, together with REX and the standard processor module hardware, for the ideal machine. The IM, in turn, is shared among the resident P-code processes and their data spaces to form multiple ideal machines.

Functionally, the system **100** is independent of the relative physical location of processes, as all interprocess communication is via logically addressed packets. Processes not requiring specific hardware environments offered by particular processor types can reside either in the host or the ideal machine space. In particular, processes that form the Level **2** system jobs are divided between these areas. Processes in the ideal space are written in extended Pascal, and each process, encoded in P-code, resides in an individual ideal machine in a general purpose processor **942, 944**. System processes that are critical to the overall dynamic performance of the system **100** are encoded directly to standard processor module machine code to gain extra efficiency. The job scheduling and allocation monitor (JSAM) task is an example of this type, and it is normally resident in an executive services processor **916, 918**. This processor contains the inter-process communications controller and, as JSAM is co-resident with the executive services subsystem **904** device manager when hosted in this processor, the reduced communication line between the two produces an increase in efficiency.

FAULT TOLERANCE

A. Introduction

The rapidly growing volume and importance of information data requiring processed by computer systems and the attendant sensitivity of users to system downtime was a major consideration in the system **100** design. Failsafe hardware with a highly fault-tolerant software system was a design requirement. Added emphasis for system availability comes from real-time (non-interruptable) operations with the always present need for data integrity.

As the focal point for large volumes of independent real-time information processing and transfers the system **100** assumes a central role in the dynamic interchange of information among users. It must be continuously available with no down-time for processor failure, maintenance or reconfiguration. To this end, the system **100** has an up-time design goal of 99.997%, operating 24-hours a day, 365 days per year.

However intrinsically reliable hardware or software elements are, failures will occur. Such failures can take many forms, from component failure due to natural aging, to sofware failure due to unforeseen transient situations. Fundamental to the design of the system **100** is the ability to continue operation in spite of such failures. The system **100** achieves uncompromising levels of system availability through its unique, designed-in capacity to survive failure. This is not achieved through the expensive and inefficient expedient of a fully redundant "hot-standby". The distributed nature of system **100** hardware and software architecture allows the system to automatically adapt around failed areas, while being able to employ all resources still available.

Neither removal of failed components, nor the addition of serviceable hardware, whose serviceability is directly verified by the system, requires system downtime. Failed processors and peripherals can be removed from and replaced into system **100** without affecting its ongoing operation. Replaced processors are automatically integrated into the system's pool of available re-

sources. In a similar way new, improved or corrected software can be introduced into an operating system **100** without requiring system downtime. In brief, a system **100** can be reconfigured and extended in both hardware and software without any affect on its ongoing operation.

All application software can make use of the system **100**'s failure recovery mechanisms and redundant hardware to achieve whatever level of failure response is dictated by the applications.

B. Failure Characteristics and Effects

No system can provide 100% availability. Regardless of the approach to fault detection, endurance and recovery, a set of circumstances may arise that the system cannot survive. For example, a common method of increasing operational availability is to have a duplicated hot-standby system, that is, a system kept up-to-date by tghe active system so it can quickly take over in the event of failure. However, during the time required for takeover, normally in the range of seconds or minutes, neither system is operational. Moreover, if the standby system itself fails before the main system is again operational, total failure occurs. The probability of both systems failing may be remote, but is always real.

A system fails when one or more components fail in such a way that the system operational capacity falls below designed minima. A system has not failed, that is, it remains available, so long as design minima are exceeded. For so long as this condition exists, the total system has not failed, whether or not one or more component failures have occurred. System availability can be improved by the following:

(1) Reducing the frequency of failures in system components.

(2) Increasing the number of concurrent failures for which the system can compensate.

(3) Reducing the time required to repair or replace a failed component.

In the system **100**, all three approaches have been used. First, system availability is increased by increasing the mean-time-between-failure (MTBF) of the system hardware and software components. The use of proven technology, automatic testing, burn-in procedures and stringent quality control contributes toward low hardware failure rates in the system **100**. Typically, after burn-in, hardware component MTBF remains very low for an extended period of time, with a rise ultimately occurring due to natural aging processes. Software components on the other hand, become more reliable with use, as all of the various states the program can enter are exercised and any incipient errors are detected, isolated and corrected.

It is generally true that if system software components are resident and extensively used this insures high software reliability, whereas new or infrequently used application components typically have a higher failure rate. To protect thr system **100** against such failures, the "ideal" machines in which the application processes execute are hardware protected environments, which confine the effects of failure to the processes in the "ideal" machines. Applications for the system **100** are written in a structured, high-level, self-checking systems programming language that generates code limited in scope to the ideal machine in which it will execute with hardware "fences" rigidly prescribing the execution arena for the software.

As a further aid in reducing application software errors, a large number of frequently exercised functions are embedded in the system **100** software to minimize the amount of application code required. This combination of features aids in achieving high system software reliability and materially eliminates the possibility of application software corrupting the system.

Second, availability of the system can be further increased by designing it to be tolerant of certain sets of faults. In the system **100** any singular hardware or software failure can be survived by the system. Also, most combinations of double failures and many sets of multiple failures can be tolerated, depending on the relative sites of the failures within the system.

Third, system susceptibility to failure can be lessened by reducing the time during which any single failure remains undetected in the system. In the system **100** extensive and frequency hardware and software checking provides assurance that the occurrence of a failure is detected quickly. Once detected, failures are immediately isolated to replaceable modules, reported, and diagnostic tasks are initiated. If the fault proves to be hardware, the offending element is logically removed from the system. The nature of the fault is displayed, and maintenance personnel can effect removal and replacement without disturbing other system operations. If a transient or software fault occurs, all pertinent data is recorded for subsequent disgnostic analysis. The failed element is automatically returned to service subsequent to passing diagnostic tests. In either case, actual repair (replacement or return to service) seldom requires more than a few minutes, and never requires system shutdown.

The meaning of the term "system failure" must be carefully defined when considering the Delta System and the concept of fault tolerance. Normally, large numbers of peripheral devices and channels are attached to the system **100**. If a device, channel, line to the processor, or interface fails, the user of the malfunctioning device perceives that the system has failed. What has happened, more precisely, is that overall system capability has been reduced although the system as a whole is still available to the surviving users. Various capabilities within the system **100** facilitate isolation of and switching around such failures.

If a failure occurs in a processor, memory, bus, or disk within the system **100**, the system as a whole will continue to operate. A degree of performance degradation may occur, depending upon the particular failure, the system configuration, and the level of activity at the time of failure. For example, if the system **100** has four general purpose processors **942, 944**, each of which is 50% loaded, and a single general purpose processor fails, the processes that were running in the failed processor will be rescheduled in the three survivors. Thus the three remaining general purpose processors will be 66% loaded. This may change slightly the characteristics and response times of the system during the time the failed general purpose processor is being replaced. However, if the general purpose processors had been 75% loaded, the survivors would end up with 100% loads. This would certainly be noticeable in the resulting queue sizes and response times.

As the system **100** is configured into distinct subsystems, each containing multiple processors, it is possible that more than one processor may fail, either in different subsystems, or in the same subsystem. The performance of the system under these multiple-failure situa-

tions (the probability of which is small) would be a complex function of processor type and loading levels.

Although the system **100** uses redundant capacity to survive failure, the utilization of that capacity fars exceeds simple hot-standby systems. Moreover, in the event of a system **100** component failure, the processes on the surviving processors are not interrupted. In contrast, for hot-standby schemes, all processes are suspended when the failure is detected, and remain suspended until takeover has been completed.

C. Fault Tolerance Capacity

Referring now to FIG. **9**, there are a minimum of two processors in any given subsystem with sufficient capacity for the survivor(s) to take on the load of any one failed processor at the designed peak system load. Each intelligent bus **912, 914** is separately managed and both are active (that is, are used for interprocessor communication). In the event of failure of a bus **912, 914**, or its controlling processor **916, 918**, the survivor provides a single bus service adequate for the design-specified peak system capacity.

As shown in FIG. **7**, within the interactive services subsystem **252** peripheral devices attach through dual-port controllers to each U and V bus **704, 708**. The primary bus for each device is preassigned by system software. In the event of either bus or ISS executive processor **702, 706** failing, the surviving bus and bus processor take over and provide a single bus service. If the device controllers fails, all devices attached to the device controller are lost to the system. However, the failed controller can be removed and replaced without interrupting operation of other controllers on the bus.

Referring to FIG. **4**, allocation of slots on each real-time TDM bus **414, 416**, is performed dynamically so as to share the load. The allocated capacity of two buses never exceeds the slot capacity of a single bus. Real-time channels operate over an independent TDM bus pair. Each bus is identical so that if one fails, the real-time subsystem **230** (RTS) can operate using the survivor. The real-time subsystem **230** will re-route data past failed external transfer switches **402, 404** (ETS) and internal transfer switches **606** (ITS).

Failure of a disk controller (that is, a file services processor **934, 936** (FSP)), a disk drive unit, or the volume mounted on it, can be survived. Critical datasets are automatically maintained in duplicate on two volumes controlled from two different file services processors **934, 936**. With dual-port disk drive units, a single data set can be accessed if the file services processor controlling the volume fails. After a period in which a duplicated dataset pair loses data synchronization due to failure of one of the pair, the "lagging" dataset is automatically resynchronized by the survivor when either a replacement or repaired unit is returned to the system.

D. Failure Detection and Recovery

The Delta System does not use a single, all-purpose failure recovery mechanism. This would be as inefficient as trying to use a single processor type in all of the various subsystems of the system **100**. A variety of methods are used for the detection of, and recovery from, failure. Each technique is best suited to the type of failure expected, and the form of recovery required. However, a set of characteristics common to all techniques can be identified to illustrate the general approach to fault tolerance adopted in the system **100**. This approach stresses two key elements: insurance against failure and an appropriate sequence of events to be triggered in case of failure.

The system **100** is configured with sufficient redundant capacity to survive failure. How that capacity is utilized depends upon the processes that use it. For example, when an application process is initiated by the system, it can be declared as recoverable or nonrecoverable. If the processor in which it resides fails, and it has been declared as nonrecoverable, no attempt will be made by the system to relocate and restart that process in alternative resources. However, if it has been declared as recoverable by restart, it will be rescheduled into suitable resources and restarted at a specified recovery entry point. (It might be illogical, in some instances, to restart a process. For example, if the process' function were only to manage a device attached to the processor that failed, there would be no point in restarting that process somewhere else). Alternately, a recoverable critical process can invoke a hot-standby process which will automatically take over if the primary process fails.

To merely restart a process in the event of failure is of no real help unless the new process can take over from the point at which the failed process terminated. What may be required in the restarted process is the overall operating context of the failed process; the values of variables, tables etc., that existed in the original process prior to failure. These values are unavailable, of course, once the processor hosting the process fails. This dictates that a process requiring extensive contextual data in order to restart must, periodically, checkpoint information regarding its current state. The action of checkpointing is similar to paying insurance. A process that wants to survive failure must accept the checkpointing task overhead. The more often the checkpointing occurs, the more transparent will be the takeover in the event of failure, but the less time is left for the original process to perform its mainline task. It is a matter of degree, determined by the relative importance of the task and the size of its variable set. If, as is frequently the case, the quantity of checkpointed data required is on the order of that required originally to initiate the process, the system itself is capable of holding each successive context set. Then, if failure occures, a newly-created successor process is passed the last checkpoint data by the system as its startup parameters. This allows the new process to take over the functions of the deceased process in a logical manner.

The basic checkpoint service provided by the system **100** is limited to retention of 16 bytes of a single checkpoint packet on behalf of a recoverable process. Each checkpoint packet sent to the system, that is, to the job scheduling and monitor (JSAM) subsystem, overwrites the previous checkpoint. The last checkpoint sent is available to the recovered process. Note that when a process is restarted in an alternative resource, the recovered process has the same job number as the failed process and therefore still owns any devices or datasets owned and used by the failed process, together with any nodes created by the failed process. Dispatched from the restart entry point, the recovered process can use the last checkpoint packet to determine a logical restart sequence.

In a job that is recognized as a network of distributed processes dedicated to the job, automatic recovery is available only to the primary node of the network. Failure of a subordinate node is reported to the node that initiated it, which can then determine what action is

required (that is, request that it be restarted, or withhold such request if restart is illogical).

Automatic restart and system checkpoint are the typical mechanisms used by application processes running in ideal machines. For example, a general purpose processor might be hosting a number of program development systems (PDS). To the terminal user of a program development system, processor failure in this instance would result in the loss of the current operation, a return to the command level of the program development system, and the generation of a system message explaining the change. At this time the user can restart his program on alternate resources. This assumes that the necessary resources are available for assignment to the re-started job. These recovery mechanisms represent a minimum overhead method for recovering from processor failure.

In critical processes, a checkpointed restart may not be acceptable if the failure of the critical process must result in a takeover by a replacement that is transparent to the rest of the system. The system software processes themselves are the prime examples of such critical processes. The replacement process must be fully informed as to the last current state of the failed process. It would be totally inefficient to attempt to checkpoint to the system the current state of a complexc process—the amount of data space required for checkpointing by the job scheduling allocation and monitoring system (JSAM) would be prohibitive. A suitable approach is dual cooperating processing. (Non-critical processes can make use of a disk dataset as a link between failed and recovered processes for large checkpoint passing.)

For critical processes, the alternative to checkpointed restart is the use of dual cooperating processes. In this approach, the critical process invokes an active backup in a separate processor. If one processor fails, the survivor takes over on behalf of the failed process. During their common lifetime, the cooperating processes maintain identical tables by various methods of communication. When one fails, takeover by the survivor is fast. The system informs the survivor of the failure and automatically reroutes all relevant packets. Within this approach, a number of different implementation schemes are possible, each with its own attributes. These schemes are as follows: (1) master/slave hot-standby, (2) hot-standby pair, (3) load-shared primary and secondary, and (4) dual parallel processes.

The characteristics and use of each mode of implementation are outlined in Section E below. Each is applied where most appropriate in the Delta System.

Cooperating processes require far more self-management by the members involved. The various schemes are set up and maintained by the processes themselves, using basic system mamangement and communication services. This high "insurance overhead" has to be traded off against the requirement for uninterrupted availability and integrity of the process within the system.

When failure occurs, the action of the system follows a defined set of steps, namely;

1. Detect that a failure has occurred.
2. Protect the rest of the system from failure.
3. Endure the failure until repaired.
4. Isolate the failed component or element.
5. Repair or replace the failed component or element.
6. Recover the system to its state before failure.

An undetected failure within the system could rapidly cause loss of overall system integraity and eventual logical system failure. For example, an undetected bad bit in a file services processor's buffer memory could cause corruption of data being passed to the disk, leading to catastrophic results some time in the future. Whenever possible in the system **100**, hardware error detection is used to immediately invoke action.

The following hardware error checks and indicators are provided in the system **100**:

For each processor:

(1) Program memory parity error.
(2) Scratchpad memory parity error.
(3) Data memory error correction occurrence.
(4) Data memory double (uncorrectable) error.
(5) Watchdog timer error.
(6) Link stack overflow or underflow.
(7) Power-fail/restart.
(8) Interactive bus parity error.
(9) Executive bus parity error.
(10) Failure to accept a packet.
(11) Physical removal of a processor.

For extensions, where applicable:

(1) Failure of real time bus synchronization.
(2) Real-time bus parity error.
(3) Transfer switch failure.
(4) Real-time serial link failure.
(5) Disk read/write cyclic redundance check (CRC) error.
(6) Disk read positioning error.

For peripherals:

(1) Reports of errors in peripheral operations.
(2) Telephone subsystem line group controllers and line interface boards.
(3) Magnetic tape controller.
(4) Loss of synchronization on a data link.
(5) Protocol timeouts.

Software errors are potentially the most dangerous to the system. A straightforward corruption of code will quickly cause a process trap. If the software (or REX) detects any illogical condition not directly attributable to an external source, or that an automatically detected failure has occurred, the host processor is forced to trap. Failed software can be indirectly detected either by the watchdog timer firing, indicating an unserviced event, or by other processes externally detecting illogical or illegal operations. What is difficult to detect is a transient, but not fatal, malfunction of a process, or a seemingly correct process behaving illogically. JSAM, via responses to its frequent status requests, monitors these transient invalid occurrences and traps the offending process if the frequency of such occurrences exceeds established system thresholds. It must be emphasized that failures of this type contribute to the learning curve effect on software reliability, in that once the cause is identified and ocrrected it can no longer contribute to system glitches. Additional protectors against invalid transient occurences exist as a part of built-in system checks, and periodically involved diagnostic utilities which reduce the probability of propogating transient errors.

If a processor fails totally, it cannot contribute to the error-detection mechanism except by failing to respond to external stimuli. Frequently, at regular intervals, JSAM polls each attached processor for a status report. Failure to acknowledge indicates an inoperative processor. Similarly, failure of a processor to accept or respond to packets from any source indicates malfunction. Either condition forces JSAM to logically remove the

processor from the system and re-route all packets to baskup processes if such backup processes are available.

Relative to a given processor, a hierarchical detection system exists. A processor is responsible for its resident processes and the hardware and peripherals associated with it. The processor itself is indirectly tested as other processors in the system attempt to send packets to it. All processors, in turn, are monitored by the primary JSAM. At the same time, a duplicated slave JSAM monitors the actions of primary JSAM. Its own host processor is itself tested by the primary JSAM. Failure of any processor is reported to or directly detected by the primary JSAM. If the primary fails the processor hosting the primary JSAM switches control to the slave (backup) JSAM. Failure of the backup is endured by the primary until the executive service processor **916, 918** hosting the backup is replaced.

Following a processor-detected fault a hardware trap is forced. This logically removes the processor from the system, thereby protecting the system from any harm that could be generated from a suspect processor. Depending on the nature of the faulty conditions which forced the hardware trap, the sytem can:

1. Alert maintenance personnel to physucally remove and replace or repair the trapped processor.

2. Run diagnostics in the faulty processor which, if successfully executed, permits JSAM to reincorporate the processor into the system.

Not all faults cause traps. For example, if the real-time subsystems **902** (RTS) detects a failed external transfer switch **402** (ETS), the real-time subsystem **902** processors logically remove the external transfer switch **402** from the system and rearrange the circuits and channels through another external transfer switch. However, if a real-time executive **406, 408** detects that the TDM bus it controls has filed, the real-time executive (together with its bus) traps.

When a processor enters the trapped state, the following sequence of events occur:

1. Normal processing is halted. All memory is preserved and only the trap handler is permitted to execute.

2. All communication with other processors is broken, except for transfer of status and diagnostic packets.

3. The CPU's registers and subroutine linkage stack are saved in scrachpad memory.

4. JSAM is informed that the processor has entered a trapped state and logically removed it from the system.

5. Copies of the Resident Executive's area of scratchpad memory and table associated with that processor are logged to SYSMON and SYSLOG.

6. If a system programmer's terminal is attached and the processor has been reset to test mode, the processor enters an interruptable idle loop, allowing the system programmer to examine the processor. Otherwise the processor automatically self-initiates power-up thereby effecting a restart beginning with physical initialization and start-up diagnostics.

If a trapped processor passes start-up diagnostics, it can be assumed that the cause of failure was either a software or hardware transient fault. JSAM notes the time of restart so that, if the failure rate of a processor exceeds a pre-determined threshold, JSAM will logically remove the processor from the system.

A totally failed processor is logically deleted from the system and reported to the system monitor for removal. A processor with no power can reside on a bus without affecting bus traffic. Physical addition or removal of a processor is reported to JSAM by the executive services subsystem **904** (ESS). Removal is equivalent to failure. Addition increases the resources known to JSAM.

A process that generates illogical or incorrect packets, but whose host processor responds correctly to JSAM status requests and does not trap, is a rogue process. The system has the following protection and detection methods to survive such occurrences.

(1) A packet has a highly structured address format. An attempt to send a packet with an incorrectly formatted address will be unsuccessful, and the originating process will be reported to JSAM by the executive services subsystem **704**.

(2) An attempt to send a packet to a non-existing address or job will also be rejected and reported.

(3) If a correctly address packet contains illogical data (for example, a wrong function code or data field), it is rejected by the receiving process. Processes use a table-decode to act upon incoming packets. A basic rule is that if it does not decode, it is rejected.

(4) The executive services subsystem **904** eventually rejects packets from a process that sends a stream of identical packets, and the executive services subsystem **904** reports the process to the System Monitor. This is behavior indicative of a process that has entered an endless loop. In such cases, JSAM will attempt to trap the host of the offending process and, failing that, will logically remove the host from the system.

A rogue process may result from insufficient implementations, testing, or intermittant hardware malfunction.

A trapped or failed processor removes from the system the processes that were hosted by it. Whatever insurance schemes were adopted by the failed processes then come into effect.

As previously described, JSAM maintains a list of processes active in each processor. When a processor failure is reported, it inspects the associated list. Any subordinate node processes wholly owned by a job are reported by JSAM as failed to each originating process. In this situation it is the responsibility of the original process to request of JSAM that they be recovered and to supply any startup parameters. The list entry for any primary node process (that is, a process initiated in response to an original create job request), is inspected to determine if the node is flagged as a non-recoverable, recoverable, or cooperating process.

If the process is recoverable, an attempt is made to find a new host with sufficient resources to accept an invocation of a replacement process. If this cannot be achieved, (of if the process is designated as non-recoverable), the job is killed and reported. Normally, however, the replacement process is initiated in a different processor and passed the last checkpoint data as its startup parameters. The new process will have the same job number as the failed process, but a different Logical Bus ID, as it has moved to a new location. A startup task is to communicate this new ID to associated prfocesses (especially device handlers of owned devices).

Examples of processes that adopt this recovery method are the program development system running in an ideal machine, and the data switch and event notification (DSEN) process that is part of the real-time subsystem **902**. DSEN is a singular process that resides in one of the real-time executive **406, 408** (RTEs) and handles the supervisory control channels to the Telephone Room Subsystem **206**. It does not maintain any

tables, and as such can be automatically recovered if its host RTE fails. It would be restarted in the surviving RTE by JSAM.

If the JSAM process list indicates that a process in a failed host was a cooperating process, the recovery schemes are supported by the system, both when both processes are present together, and also after one has failed. This is accomplished in the following way:

(1) Each process in the system is identified by a Process ID (PID), a 20-bit identification in the system. Seven bits of the PID identify the processor that is hosting the process. This is referred to as the Logical Bus ID(LBID) of the processor, and is a number in the range 64 to 127. The LBID is assigned to the processor when it is integrated into the system at power-up. When a processors traps, its LBID is removed from the system. This eliminates the possibility of continuing to route packets to a trapped processor. The processor is assigned a new LBID when it is put back into system operation. This also makes clear that a process ID is variable, depending upon which processor is assigned to when initiated.

(2) Major software components require invariant global addresses in the system, irrespective of the processor in which they reside. A processor that hosts a critical process (for example, the System Directory Manager) is also assigned a Bus ID (in the range 0–63) that is fixed for the particular software process it is hosting. This means that any process wishing to communicate with, say, SYSDIR will always use the same 7-bit Bux ID, irrespective of the actual processor in which SYSDIR resides. Such an ID is termed a System Bus ID (SBID). The sender uses the SBID or LBID in the packet header, and the executive services subsystem **904** maps SBIDs and LBIDs to Physical Bus Ids (PBIDs) during packet transfer. If a processor hosts more than one system process, then more than one SBID may map to the same processor.

(3) Processes that use cooperating process recovery schemes are also major software components, as they are critical to system performance. SBIDs are therefore normally assigned in sequential pairs, with the first SBID relating to the processor that hosts the first of the cooperating process pair, and the subsequent SBID pointing to the processor hosting the second.

(4) JSAM and the executive services subsystem **904** use the SBID pair to advantage when handling packets for duplicated processes. A "Use Code" associated with the SBID pair tells the interprocess communication subsystem how to translate an SBID to a PBID when one of the pair fails.

On failure of one of a cooperating pair, the following sequence of events occur:

1. The surviving process, if originally primary, remains primary and packets continue to be assigned to it.

2. If the surviving process is the backup process it is now designated as primary and packets are rerouted to it.

3. If the cooperating process is designated as recoverable and resources are available, a new process is invoked and designated as the new backup. The surviving primary then transfers its tables to the new backup and and normal operation continues.

4. If adequate resources are not available, operations continues with the surviving primary until the failed procdssor is replaced and reports itself to the system. Action is then as in (3). This is the normal sequence for failed executive processors (that is, processors controlling one of a bus pair) as only two of each exist in the system, each one hosting a member of a cooperating pair.

Note that this scheme works equally well in maintaining duplicated datasets as well as duplicate processes. When a duplicated dataset is opened, the two file services processors **934**, **936** which are attached to the volumes containing the two copies are assigned an SBID pair for the purpose of subsequent record transfers. This allows the SBID use code handling to be taken advantage of in the event of failure.

A failed processor which cannot communicate or which has been logically isolated by JSAM must obviously be removed to a test station for repair. However, for any failure, including processors which subsequently pass diagnostics and are reassimilated, trap information is available in the system log and system monitor. This trap information contains the memory and register dumps sent from the processor coincident with the failure, and the log packets from it and from other processors reporting operation, status and problems before and during failure.

A trapped processor offers considerable scope for both passive and active diagnostics to isolate the cause of the failure. Following a trap, a dump of selected portions of memory is made to the log, which is then available for analysis. The processor can be flagged to inhibit automatic restart, thus the condition of the processor's registers and link stack are available for inspection, either directly by requests from a console or terminal to the trapped processor or indirectly from the dump. The processor's ports and connection to the bus can be tested by special diagnostics initiated from a console or terminal.

When the processor is restarted, either automatically after the trap, or if flagged, after completion of diagnostics, the restart sequence is essentially identical to that of a processor newly-added to a Delta System:

(1) Physical Initialization—Checks the various memories in the processor and reinitializes all of the processor variables, flags, ports and scratchpad registers. The extension board of the standard processor module also performs a physical initialization to quiesce any attached bus or devices.

(2) Start-Up Diagnostics—At the end of a physical initialization the processor reports to JSAM. JSAM then normally requests that REX load and execute the startup diagnostic programs. These diagnostics verify the operation of various processor functions, registers, and self-checks such as parity and ECC. A detected failure causes the processor to be inhibited from reassimilation into the system.

(3) Logical Initialization—After the processor reports successful completion of diagnostics, JSAM requests that a processor type-dependent initialization program be run in the processor. Upon notification of completion JSAM inspects its system resource file to determine if this processor is currently required to host any system processes. If so, REX is notified to load and initiate the processes. This could include a backup process to reform a pair of cooperating processes after a period when the surviving primary has been operating alone. When the new backup is running, the primary is informed and proceeds to bring the backup up-to-date by table transfers (or record transfers if a duplicated dataset). The system is then in a position identical to that which existed before the initial fault occurred.

E. Cooperating Processes

In adopting a dual cooperating process approach to sustained system functioning, several implementation schemes have been used. Each has specific advantages when used in the various subsystems and functional areas of the Delta System. All system software processes and some utilizies and handlers use dual processes to guarantee uninterrupted system operation in the face of processor or bus failure. The following sections describe four specific schemes used in the system **100**. The two aspects which characterize each particular scheme are: (1) the relationship between each process during normal system operation (that is, when both processes exist) and (2) the reaction to failure to one of the process pair.

During normal operation of a master slave hot-standby pair of processes the user communicates only with the master process. The master is responsible for updating the slave's tables to maintain its takeover capability. If the master fails, the slave becomes the new master and all subsequent interaction is between the new master and the user. If either the master or the slave fails it is reported. When the failed process is restored, it becomes the slave to the surviving process.

This scheme is used for JSAM. One of the two executive services processors **916, 918** hosts the prime (master) copy and the other hosts the backup (slave) copy. The fully redundant master/slave hot-standby is used because JSAM is the major fault-recovery mechanism for the rest of the system, and must have a fast response backup to recover.

During normal operation of a hog-stanby pair, the user communicates with the prime process which acts upon the request. The backup maintains the same information as the prime, but does not act upon requests. If the prime fails, JSAM notifies the backup that is is now prime and it takes over the process to complete the request.

This scheme is used mainly by system processes resident in ideal machines (Pascal-sourced processes running in a general purpose processor). Although the backup is fully redundant, it is tying up only an ideal machine, not a whole processor. As each general purpose processor can host a large number of ideal machines, this scheme is not inefficient in machine usage terms, but is highly efficient in terms of the reduction of interprocessor data-exchange traffic. Due to the large amounts of memory available in each general purpose processor (up to 8 Mbytes), system processes tend to build up large memory-resident data bases, which are faster and more easily maintained than disk-resident datasets. It would be very inefficient to have to pass this data between a prime and a backup as in the master/-slave scheme. It is far more efficient to let both the prime and backup processes dynamically maintain them by parallel processing.

Examples of system processes using hot-standby pairs are SYSDEV (System Device Manager) and SYSDLO (System Device Long-on Process).

A load-shared primary and secondary scheme is used where dynamic load sharing is required between the two processors. Each is assigned work by the prime, based upon some specific load-sharing algorithm. When one of the team fails, the survivor takes on the load of the failed member. Obviously, the load assumed by either member can be no more than 50% of total capacity so that in the event of failure adequate capacity exists for takeover, but the two lightly loaded members, during normal operation, can provide better response time than if a hog-standby scheme had been used. If the pair is also driving duplicated hardware resources, this scheme also shares the load on the hardware, thus improving overall system performance.

This scheme is used in the real-time executives **406, 408** of the real-time subsystem **230** for the processes that control circuit switching and supervision of the real-time bus pair. This results in equal loading of time slot assignments on the bus pair such that the task of reconstituting a failed bus interconnect is not biased depending on which bus fails. To speed the response of takeover, each real-time executive **406, 408** can directly monitor the bus activity of the other real-time executive, and can immediately detect if the other has failed (by detecting loss of the bus clock). In this event, the survivor immediately begins the takeover sequence, which must be completed very quickly in a real-time environment.

A scheme using dual parallel processes can be used when there is a static assignment of requests to copies of the same process in different processors. The user is directed to talk to one or another of the processes, depending upon his particular request and how the request maps onto the preassigned responsibilities of each process in the pair. If one fails, however, the survivor takes on the load. During normal operation, each member of the pair ensures that the other's tables are up-to-date such that a takeover can occur.

This scheme is used with device handlers in the two interactive service executives **702, 706** (SXs). During normal operation each interactive services executive is responsible for a subset of the devices attached to the U-V bus pair **704, 708**. Each is host to copies of the various device handlers required. When a user wishes to communicate with a specific device, he passes a request to SYSDEV. SYSDEV acquires the requested device (if free) from either the primary or secondary interactive services processor, depending upon a predefined load-sharing plan. It is possible for a user to be talking simultaneously to one interactive services processor for one device and the other interactive services processor for a physically different, but same type, device.

The comments on loading and response time for load sharing discussed previously apply equally as well to this scheme (that is, 50% loading at maximum, but faster response and better hardware utilization).

The various techniques for achieving high integrity with datasets are summarized below. A dataset can be created as a duplicated dataset. This allows a prime copy of the dataset to exist on a disk drive attached to one file services processor **934** (FSP) and a backup on a drive attached to a different file services processor **936**. During the sequence for control records, the record is sent to the prime file services processor **934** which updates one copy of the file. The prime file services processor **934** passes the record to the backup file services processor **936**, which updates the second copy, and responds to the user. In this way the user is assured of successful updating of both copies. If one file services processor **934, 938** fails, or the drive it controls falls, or the vcolume containing the dataset is scratched, subsequent records are sent to the surviving dataset, (which is automatically prime) and changes to the dataset are marked. When the failed dataset comes back on-line, the two datasets are automatically resynchronized by copying changed records from the survivor to the lagging dataset.

PRINCIPLES OF OPERATION

A. Operational Principles

Underlying the functional organization of software components of the system **100** is a general operational structure that is consistent in any processor of the system **100** and applies to all levels of software functionality. Major system components, applications, utilities and ideal machine monitors are all implemented using the same set of basic techniques thus enforcing a rational method of building both the system and the application tasks.

Three major concepts are fundamental to all system **100** software. These are:

1. The Process—An identifiable separate element consisting of a prgram which is resident and confined to a single processor in the system. Every processor in the system **100** can host a large number of processes, where each process can be an element of some larger functional component.

2. The Job—A unit of work composed of a logical group of processes created expressly for a purpose. The component processes of a job can individually reside in any of the processors of system **100**, their allocation to a processor determined by their resource requirements and availability rather than any "a priori" assignment. The system **100** supports a large number of concurrent jobs. Jobs are created and terminated dynamically in response to internal and external events.

3. The Packet—Each job is a cooperating set of system-wide processes that intercommunicate by way of discrete packets. A packet is a 16-word data structure that any process can send to any other process, provided the identification of the destination process is known. The transfer of packets between processes is supported at the basic hardware and firmware level by the buses, processors and resident executive of the system **100**.

The use of discrete packets to communicate between the elements of a job formalizes cooperation and ensures high job reliability. Processes, jobs and packets are described in detail in the following sections.

Referring now to the real-time executive **408** shown in FIGS. **45** and **6** by way of example, each processor comprises a CPU **504**, scratchpad **510**, program memory **506**, **508**, **614** and data memory **616**, a firmware resident executive **508** and dual port access to the interprocessor bus through a port control unit **502**. Each processor type can be extended with one of five normal extension boards (NEBs) that can extend the amount of program and data memory, provide interfaces and channels to attached devices or buses, and extend the instruction set of the CPU.

A process is the basic component of all user and system activities in the system **100**. Each process resides in a single hardware processor and performs work in the system by using the resources of the processor that hosts it. Associated with each process is some executable program code resident in the processor, together with a set of data uniquely associated with the process, called its "context", and a control data structure called a process control block. Each process requests of the processor hosting it the computational and other resources required for execution. At any one time, a large number of processes may be competing to use the processor. Others may be dormant awaiting occurrence of some internal or external event. One function of the processor's resident executive (REX) is to manage the allocation of its processing resources to each process as the process become eligible for execution.

At any one given time, a processor is performing work on behalf of only one specific process. When a process is initiated, the processor's internal registers are loaded with data describing the initial state of the process, and the machine proceeds to execute the program code associated with that process. The processor remains dedicated to executing that program until control is either voluntarily relinquished back to REX or some external hardware interrupt occurs. The state of the process when suspended, as described by the internal machine registers, is saved in the process control block (PCB). The process control block is enqueued to either await an event or to complete once again for processor resources. A certain area of the processor's scratchpad memory that was assigned to the process when it was created (its scratchpad context), and used by the process when it ran, is also left untouched until the process is allowed to execute again. The processor is assigned by REX to the "context" of the next priority process (by loading its desired register values, and pointing to its private area of scratchpad). In this way, a previously suspended process resumes execution, or a newly created process begins its execution.

In uniprocessor (i.e., single cpu) machines, this technique of sharing the processing resources of the machine among a number of processes is termed concurrent process operation or multi-processing, although processes actually execute in a one-at-a-time sequence. Concurrent process operation refers to a number of processes resident in the machine which are all concurrently at some point of progress, with each one being singularly advanced through its lifetime as it is allowed to use the machine's resources. In the system **100**, simultaneous processing does occur as processes in physically different processors execute in parallel.

The difference between a process and a program must be emphasized. A program is a static set of code that is loaded into a processor, and used when executing a process. A program is still a program even when it is resident in a library on a disk. A process is a dynamic entity that exists only in the processor with a processor control block and a data context allocation. A process uses a program to achieve its objectives. It is possible for a number of like processes to use the same loaded copy of program code in the same processor. When one process suspends running at some point in a program, another process may begin operating at a different point in the same set of program code. What distinguishes one process from another is its unique scratchpad memory, data context and process control block which collectively describe the process and its current state.

Every process that exists (i.e., has been created) in the system **100** has two important identities.

Firat, each process is uniquely identified by Process Identification (PID). This is a 20-bit code that locates a specific process in the system (including both processor and context). When individual processes communicate with each other, the Process Identifications are used to locate the destinations. Although the system **100** is constructed from a number of separate hardware processors, it is transparent at the individual process level. A process can identify and communicate with any other process, whether resident in the same or a different processor location, with equal ease by way of the Process Identification.

A second identity that a process has in the system **100** is the job number with which it is associated. This is an organizational identity that is given to the initial process when the job was created and further extended to apply to any other processes created for the express purpose of completing the job. A job is a complete, individual functional entity created to perform a specific task. To perform its task, more than one process may be created in the various processors of the system **100** to work together to implement the job. Each job is assigned a unique 16-bit job number, and each constituent process is identified as being a member of the job by the same 16-bit number.

Processes communicate via packets of 16 words each. Packets are of two basic types—process and data. Process packets are used to synchronize operations, invoke functions, request services, report status and respond to any of these. Data packets are used to transfer blocks of data from one process to another.

The sending process is responsible for packet construction and format. Each packet begins with a header consisting of the Process Identification of the target processor and sendin process plus function codes defined and used by the target process and sending process.

In data packets, the header is followed by one or more bytes of control information and up to 28 bytes of data.

In process packets, the header is expanded to include the Process Identification of the source, thus facilitating a reply from the target to the sending process. Most process packets also include a job number or other validation number, plus additional information as required by the receiving or subsequent processes.

A process sends a packet by invoking a REX function, directly for privileged processes and via an ideal machine function or procedure for restricted processes. When a packet arrives for a process, either as an expected reply to a previously sent packet, or unsolicited, an "event" is defined to have occurred. A suspended process waiting for the packet is activated by the event and can queue with other dispatchable processes to use the processor in order to receive and interpret the packet. A packet arriving unsolicited for a running process is placed on its list of events awaiting action that is maintained on its behalf by REX.

To ease the problem of decoding incoming packets, a process can build a 16-entry branch table of alternative start addresses. Each packet header includes a 4-bit function code which can be used to index to a specific routine within the process.

Packet transfers are handled on behalf of a process by the REX in the host processor and by the Inter-Processor Communications Subsystem. The multiple REX utilities in each of Delta's processors and the executive services subsystem **904** form a location independent inter-process communications network. At the request of the sending process, REX enqueues the packet or block transmission request until either of the output ports are available. When available, REX places the packet in an output port of the processor and transfers control of the port to inter-process communications.

Inter-process communications transfers the packet out of the source host, over the high-speed executive bus **912** or **914** to the port in the processor hosting the destination process. (This could be the same processor from which it was sent.) The arrival of the packet is posted to the destination process as an event.

A job is an organization of one or more processes invoked to perform a specific task. A job is a system-wide entity that can be created or terminated by a single request, and uniquely identified during its existence from other jobs in the system by a job number. The system **100** can concurrently host a large number of different jobs.

A job consists of a set of processes, each resident in a system **100** processor, and having a tree-structured hierarchical relationship to each other. This structure is referred to as a process network with each component process termed a node of the network. Each job's process network has a single process called a primary node that is created as the initial response to a request for a job to be created.

As the primary node proces executes, it can request that other processes be created. For example, this can occur when the primary node requires the services of other system **100** resources not available in the processor in which the primary node resides, or it can occur purely as an organization expedient. Each process created by the primary node is subordinate to the primary node. Thus, if the primary node is destroyed or terminated and not restored, any nodes it created (i.e., any processes it started in a different or the same processor) are also terminated. As each process is created, its process identification is passed back to the creator; and the process identification of the creator is passed to the new process. This allows the primary node process and its subordinate node processes to communicate with each other by packet transfers. By this mechanism, a simple star network of processes can be formed.

To extend the network further, it is possible for each subordinate node to itself create further sets of subordinate nodes. The relationship between the node and its created set of subordinate nodes is the same as that which exists between the initial primary node and the first set of subordinate nodes i.e., if a node is removed then any nodes it creates are also removed. Each node can communicate with the processes of the nodes it creates or its own creator, but not with processes of other nodes, or nodes below its own subordinates (unless explicitly given their process identification numbers). Thus the initial star-structure of the job's process network is transformed into a general tree-structure. The creation of subordinate processes can be carried to anh depth, and nodes and sub-nodes can be dynamically added or removed during the lifetime of the job. The job's tree-structure is not a static organization, but can expand and contract as required. If a node is removed, all of the tree-structure below that point disappears. To terminate a completed job, all that is required is to terminate the primary node. Every process created is allocated the job number of its creating process. Thus all processes in a process network have the same job number.

A process network can be likened to the structure of a block-structured program. The main body of the program (the primary node) can declare (create) and use (send packet to) a number of procedures (subordinate processes). Each of these procedures can themselves be structured by internally declaring to it further procedures (create a second level of subordinate processes) that it can use, but which are not within the scope of the main body of the program (the primary node only knows the process identifications of the processes it creates, not those of processes created below them.)

As a process is a processor-level entity, a job is a system-level entity. The primary node and subsequent subordinate nodes are created in any processor that has sufficient resources available to host the nodes. The inter-node links are set up by exchanging process identification numbers.

In many cases, a single primary node is all that is required to realize a job. This is a specific instance of the general structure.

Jobs are initiated and processor resources are allocated by a set of processes referred to as the job scheduling and allocation monitoring (JSAM) subsystem. JSAM itself is an example of a job, and is bootstrapped into existence when the system **100** is first powered-up. One of JSAM's first actions is to initiate all of the required system jobs bringing the system **100** up to operational readiness. JSAM can then accept requests to create new jobs.

JSAM receives a request to creat a job as a single packet. As JSAM's process identification is fixed and known globally, this can be sent from any process. (Note: All level **2** system processes have fixed process identification numbers and fixed job numbers so that they can be addressed from anywhere in the system.) Included in the request is a function identification that defines the program required to be run as the primary node, a function mode, and a set of start-up parameters for the primary node. JSAM's job management routines note the existence of a new job by creating a job control block (JCB) that contains all relevant details relating to the job and is periodically updated to reflect the current job status. As the process control block is to a process, the job control block is to a job.

Job management passes the function ID to an area in JSAM referred to as node management, which uses the function ID to obtain from the system directory manager the resource requirements of the program or programs capable of fulfilling the function requirements. For each program, this includes the processor type required, the processor-specific resources required (DMA channels, for example), and any external devices required. Node management attempts to find a processor with sufficient resources to match the needs of the program. If it can find a processor not only with sufficient resources available but also with the actual required program code already loaded, that processor is the first choice. Otherwise, the first processor found that is able to host the node process is selected. Node management sends a create process request packet to the REX of that processor, which loads the program (if not already resident) and creates the process. A separate function of JSAM is to monitor and update the status and loading of all processors. Thus, JSAM can efficiently search for an acceptable host processor by internally inspecting its own processor status tables.

For any desired function, there may be more than one program or program versions capable of fulfilling the functional requirement under the following circumstances:

1. Functionally equivalent programs which must be adapted in order to conform to differences in the instruction repertoires to permit execution on different processor types.

2. More than one mode or version of a processor type or of REX may co-exist in a single system, particularly during an upgrade or retrofit. Different program versions may be necessary to properly utilize the different hardware versions.

3. A new version of a program may be introduced into a system for limited operational testing alongside the current version.

4. A new conditional version of a program can be tentatively introduced in such a way that any subsequent processor failure causes the new program to be removed from availability and replaced by the predecessor version.

Where case one or two exists, a given function ID will reference a list of one or more programs for each processor type, model or version. Normally, each list will include an operational version of the program to be utilized with the associated hardware. In case four, if the entire system is operating in the "conditional state", then a conditional version of a program will always be selected in lieu of an operational version. (The conditional state must be set manually from a system console and is automatically reset in the event of any processor failure). In case three, the create job request can designate a mode (2 through 255). If a test version corresponding to the mode exists, then it is to be used. Otherwise, the current operational (or conditional) version is to be used. The mode designation is applied to any subsequent node requests initiated under that job number.

If node management cannot successfully initiate a job's primary node, job management returns a negative acknowledgement to the requesting process. This process is responsible for effecting any subsequent retries.

Once a job's primary node is running, it can send requests directly to JSAM's node management to create subordinate nodes. Node management acts upon these requests identically to an initial request from job management for a primary node. To reduce the burden in setting up a large number of subordinate nodes, a single request can specify up to three function IDs in one request, each one for a different subordinate node. Alternatively, a single value called a node ID can be supplied that references a node descriptor table in JSAM with each entry containing up to five function IDs for subordinate nodes. If all of these nodes can be successfully initiated, node management responds to the requestor with the process identification numbers of each node process created. This can be repeated to an arbitrary depth of sub-node creations.

Apart from providing node creation services while a job is running, JSAM can provide failure recovery services in the event a processor fails. JSAM is able to do this since it is responsible for both monitoring the status of each processor attached to the executive bus **912, 914** and managing the intelligent controllers of the executive bus **912, 914** which perform all packet transfers.

If JSAM detects, or learns, that a processor has failed or trapped, it inspects its tables to identify what node processes were resident in it. If any subordinate nodes were hosted in the failed processor, the failure is reported by sending an alert packet to the highest level surviving control node which is responsible for restarting the failed node, if this is the logically correct course of action. If a primary node is detected to have failed, and it is flagged as recoverable, JSAM will attempt to reinitiate it in an alternative processor, restarting the new primary node at a predefined restart address. JSAM will pass to the reinitialized primary node the last 16 bytes of checkpoint data stored by the processor prior to the failure.

JSAM can also assist Level 1 and 2 system processes to efficiently implement one of a number of prime/-backup recovery schemes through its ability to control the routing of packets. Packets addressed to the prime version of globally-known paired system processes are automatically rerouted to the surviving member of the pair if failure occurs.

A process that originally sponsors a job request to JSAM is not ignored once a job is underway. When a process originally sends a job request to JSAM, it also provides a reference number. When the job is successfully running (or rejected if it can't be started), JSAM reports back to the requestor, quoting the reference number, the process identification and status of the primary node, together with the job number that JSAM assigned.

A job can terminate in one of three ways. It can run to completion and request the deletion of its own primary node it can fail to start after waiting its maximum delay on a schedule queue. Or, it can be forcibly terminated by a privileged process.

At the individual process level, within each processor, the time management services of REX can be used to manipulate an arbitrary number of programmed timers. With these, time-dependent operations can be organized in the processor. At the system-wide level, it is also possible to organize complete jobs on a periodic basis. A request for JSAM to create a job can be made on a time-scheduled basis rather than immediately. A scheduled job request supplies, in addition to the usual program ID and reference values, a time at which the job request is to be acted upon. The time can be scheduled in six minute intervals for a given day of the week, date and month or Julian date, and the same job can be repeatedly rescheduled according to a variety of regimes. Once the schedule time matures, the job request is handled as if just received for immediate action. For finer time control than six minutes, the primary node can use its host REX to provide timer functions.

JSAM also allows individual packets to be stored and then mailed to a specified proces at a predetermined future time (again, at intervals of minutes and reschedulable). Rather than send a packet directly to a process, a scheduled activity packet can be sent to JSAM containing a packet header, a scheduled delivery time, reschedule criteria, and up to 16 bytes of data. When the time matures, JSAM constructs a packet using the header provided (which contains sending and recieving process identification numbers) inserts the user-supplied data, and posts it to the destination.

Each system **100** process or supports two process types, main processes and subprocesses. In operation, both types are treated identically by the scheduling and dispatching functions of REX. The difference between the two is related to the hierarchy of their creation and the use of a processor's scratchpad memory.

To achieve very high speed multi-process operation, processors in the system **100** have a large, 4096-word set of working registers referred to as scratchpad memory. Each word of this memory has a read/write time compatible with the other major registers in the processor's CPU. At any point in time, the state of a process is reflected by both the contents of the CPU's internal registers, and the content of a set of working registers in scratchpad memory that the process is using. If a process is to be suspended in favor of another process, the current values of all of these locations must be preserved until the suspended process can execute again. After saving a suspended processes' "context", new values reflecting the startup state of the replacement process must be supplied. This register-swapping is an overhead in multi-process environments and has to be reduced to a minimum.

To achieve this in a system **100** processor, a processes' scratchpad register set is left untouched while it is suspended. This is achieved by assigning each main process its own private area of scratchpad memory, called its context. During the lifetime of a process, its context in scratchpad memory remains assigned for its use only. Thus, when a main process is suspended, only a small number of internal CPU registers need be saved, as the contents of its context in scratchpad will be preserved until the process runs again. This reduces the process swapping overhead to exchanging six SPU and processor extension unit registers together with a selectable depth of the subroutine link stack between the process control blocks of the processes, where they are saved while a process is suspended.

Scratchpad memory is divided into 32 pages of up to 128 registers each (128 registers equals 1 page). A context consists of a part or all of a page. Page zero of scratchpad is reserved for global use, page one is REX's own context, and pages 2–31 are available for allocation to other main processes. Each context must start on a page boundary, therefore, 30 main processes can exist, each with a context identification (CXID) equal to the page number.

A page is further divided into eight packets of 16 words each. When a main process is created, the context can consist of any number of packets from a minimum of two to a maximum of eight. By convention, the context is as small as is reasonable. All unassigned packets of scratchpad are retained by REX and are allocated dynamically to existing processes (main or subprocesses) as required. The approach to the use of the scratchpad resource is thus to initiate processes with the minimum packet allocation of context, and then request more as requires.

As JSAM continously monitors the status of all system **100** processors, it knows if a processor has any free contexts available. This is taken into account by node management when attempting to find an eligible host processor.

Before a main process can be initiated, the code of the program specified in the create process request must be resident in the processor's program memory. It it is not, REX sends a request to the system loader (part of the system directory manager SYSDIR) to load the program before creating the main process.

An initiated main process therefore consists of an allocated context in scratchpad memory, a process control block constructed in data memory, and a program in program memory. A new process is immediately placed on the list of dispatched processes to compete for use of the processor's resources. This allows it to perform its own logical initialization which could perhaps include requesting JSAM to create a set of subordinate nodes which will be allocated under the same job number.

Within each separate context allocated in scratchpad for main processes, more than one subprocess can operate. Each main process can create subprocesses all of which share the same context. These subprocesses themselves can also create further subprocesses. Up to 255 subprocesses can share the same context as a single main process. Each subprocess has a subprocess control

block identical in format to a main process control block and can compete on equal terms with a main process for use of the machines resources. Subprocesses can queue events in a manner similar to that of a main process.

Main and subprocesses introduce a hierarchical structure into process organization. Only main processes can be initiated by JSAM in response to a create node request, specifying a function ID. Once running, a main process can then create any number (to 255) of concurrent subprocesses to implement the function of the node. The program code used by the subprocess must be loaded before the subprocess is created, and is usually part of the program loaded when the main process is created.

In addition to the context ID (CXID), a subprocess has an identifier called a subprocess ID (SPID) in the range 1–255. A main process has a subprocess ID of zero. Thus any process can be uniquely referred to by its processor ID, CXID, or subprocess ID.

An example of the use of a subprocess can be found in the file service subsystem **908** (ESS) which has a main process resident in each file services processor **934, 936**. Once a dataset has been opened by a user, a READ or WRITE access request is sent to prepare buffers for a subsequent GET or PUT operation. In response to an access request, the file services subsystem **908** creates a specific subprocess to handle subsequent communications for all GET/PUT operations. Using this scheme, it is possible for a user to specify multiple READ/-WRITE access requests, and perform concurrent GETs and PUTs, each one being handled by a separate subprocess in the file services subsystem **908**. Each individual access subprocess is uniquely identified by the different subprocess ID returned to the user in response to each READ/WRITE access request.

The shared main context of scratchpad is a useful means by which main and subprocesses can communicate. To provide a non-conflicting private context area for each process any combination of the eight packets that make up a context can be defined to preserve and restore the context each time the subprocess is suspended or dispatched. In each process control block, and 8-bit field is used as a map of the context packets to be swapped in from data memory when the process executes. A related field in the process control block points to a save area in data memory to which the same packets are swapped when the process suspends. The swapped packets are referred to as the subprocesses' shadow context.

A variation of a main process exists called a bypass main process. When a main process is created it can be designated to not only receive all packets addressed to itself (i.e., subprocess ID=0), but also all packets addressed to the context regardless of the subprocess ID (i.e., subprocess ID 1 to 255). This can be used where a main process needs to monitor packets being sent to its own subprocesses.

Processes (main or sub) that are not suspended awaiting some internal or external event are maintained on a queue of dispatchable processes competing for the use of the CPU. Each processes' process control block contains a bi-level dispatching priority, consisting of a 5-level class, and within each class, a ranking from zero to 255. Therefore class zero, rank zero is the lowest possible priority in the system, and class 4, rank 255 is the highest.

REX always selects from the queue the highest priority disoatchable process for execution. Once given control, the process is allowed to execute until it voluntarily relinquishes control or is interrupted by a hardware interrupt. Typically, a process can give control back to REX when it WAITS or QUITS, or attempts to perform an operation via some unavailable resource. The process is then either terminated (i.e., after a QUIT) or placed on one of a number of event queues, each queue related to the reason the process was suspended. When the appropriate event occurs, it can then return to the dispatch queue to compete for use of the CPU.

The dynamic activation and suspension of processes is determined by the occurrence of events. An event is any change of state, either hardware or software, that can be recognized and communicated to a process.

An external hardware event can cause an interrupt to occur, causing an interrupt handling program to be entered. All interrupt handling is managed by REX and is transparent to the user's process. Dispatched processes run at the noninterrupt level **3**, with REX's interrupt handlers being able to run at interrupt levels **0, 1** and **2**.

Processes can manage their activity by way of software events. Typical software events include:

(1) Receipt of an input packet via REX.
(2) Expiration of a time interval.
(3) Completion of an I/O operation.
(4) Termination of a subprocess.
(5) Availability of a resource.
(6) A signal from a coresident process.
(7) A user-defined event.
(8) Availability of a previously unavailable resource.

An event and its relationship to a process is defined by a data structure called an event control block (ECB). When an event occurs, an event control block describing the event is linked to the processes' process control block. If the process is executing, the event control block can be actively checked by the process for occurrence. Alternatively, a process can wait for a specified event to occur, or for any event related to that process to occur. Until the event occurs, the process can be suspended and other processes can use the machine.

Events can be either unsolicited or solicited. An unsolicited event typically occurs when a packet arrives unexpectedly for a process. This will be picked up by the process either next time it checks its list of event control blcoks, or the next time it "waits" for any event. A solicited event is one which is expected by the process. For example, in a packet exchange, a confirmation packet is often sent back from a destination process to the sender. In this case, the sending process can set a timer associated with the event, and when the event actually occurs (in this example, the receipt of the return packet), the process can check to see if the event occurred before or after the timer expired.

Packets are classified by each process into 16 function code types, corresponding to the function code in the packets addressed to the process. The process defines a function code table that contains a program entry point for each permissible function code. The table entry is used as a vectored return address for the process when the event is either actively or passively detected. The function code in each packet header relates directly to this vector table. Each acceptable function code causes a hump to a specific handling area of the process when the "packet received" event control block is checked or

waited upon. Packets containing function codes not posted in the table are rejected by REX and discarded.

Also associated with the function code is the function mask. This is a 16-bit mask that designates a function code as an unsolicited event-type if the corresponding bit in the mask is set. This allows packets with incorrect pack function codes to be rejected and not be treated as significant events. This provides a degree of protection against ill-formed packets. Both the function mask and a pointer to the function table are kept in the process control block. For further decoding of unsolicited packet-related events, word 4 of the packet is normally reserved for further packet function identification. For solicited events, the process can provide a 16-bit reference value to identify which event is which when multiple events are expected.

Processes can explicitly generate future events by setting timers. Timers are set up by a call to REX from a process, which generates a timer event control block and places it on a list of similar timed events that REX manages. The process can continue execution, checking for the occurrence of the timer expiration event directly, or can suspend until the time specified in the call to REX matures. The timer period can be specified in increments of 34 microseconds, 1 millisecond or 1 second intervals, with a 16-bit value giving 65,536 increments.

Each processor in the system **100** uses its own hardware clock to generate the basic 34 microsecond time interval. The absolute clock values of all of the processors is regularly synchronized by JSAM, whose host processor clock contains the system's "master clock".

A further class of user-defined events relates to the cooperative use of resources by a number of processes and the synchronization of process activity related to the use of a resource. For example, access to a common stored data memory area by a number of processes, each of which references and updates values in the area, must be controlled such that only one process is performing an update at any one time. This can be achieved by the use of a named binary semaphore managed by REX that can be enqueued upon by processes wishing to access the single-use resource. A process successfully requesting the use of the resource sets the semaphore. Any other process requiring to access the resource first tests the semaphore. If it is set, the process is suspended until the semaphore is reset by the process currently using the resource. The next enqueued process is then dispatched and the semaphore is reset. The binary semaphore acts as a one-process "gate" to the resource. An individual semaphore is referenced by a user-supplied 16-bit number.

For access to limited multiple resources, a general semaphore service is provided by REX. This is used in a similar way to a binary semaphore except that it has an associated counter, rather than a simple go/nogo binary value. When the general semaphore is first set up by a call to REX, an initial count is provided that reflects the amount of resource available. Every access to the resource is preceded by a call to the general semaphore, which decrements the value. If any process decrements it to zero (indicating exhaustion of the resource), the calling process is suspended until another process finishes using one of the available resources and increments the value from zero. The suspended process is then dispatched. Use of general semaphores permits a number of processes to share a pool of devices (at some higher level than provided by the system device manager).

C. Inter-Process Packet Communications

Processes communicate with each other via packets. A packet can be sent from any process in the system **100** to any other process in any processor. To send a packet, a process constructs the 16-word packet in either data memory or in the processes scratchpad context and then calls a REX routine to send the packet. REX queues the packet on the executive bus output ports of the processor, and loads the packet into the first free port (X or Y bus ports). When the bus polling microprocessor (part of the main interprocessor executive) polls the part, the port responds that a packet is ready. The microprocessor reads the first word of the packet out of the port. This first word contains a field which identifies the logical address of the processor hosting the destination process to which the packet is being sent. This logical address is transformed (via a table look-up) to the physical address of the destination processor. The in-port of that processor is checked. If it is busy (still unloading a previously sent packet), the transfer request is queued by the microprocessor which continues polling other processor's out-ports.

When the destination in-port becomes free, the packet transfer is completed between the out-port of the sending processor and the in-port of the receiving processor. The parity of each word is checked during transfer. The REX of the receiving processor then transfers the packet to data memory, places the pointer to the packet in an event control block (having constructed an event control block if the packet is unsolicited) and posts the event control block to the receiving process. The receiving process can then pick up the event control block (and hence the pointer to the packet) when it next checks its list of event control blocks or "waits". The function code in the packet header is used to vector the receiving process to a service area for that packet type, using the predeclared function code table.

During a transfer, the process that sent the packet has a number of options. The simplest is to send the packet and continue. However, for security, most interprocess packet eschanges are in pairs, with the sending process expecting some form of response from the receiver, usually either a reply to the packet or an acknowledgement of receipt. To facilitate this during the send operation, the sender can specify a return event control block by which it can be informed if a return packet related to this send operation is received, together with a time-out value defining a maximum response time. At the receiving end, the destination process can use the REX "RESPOND" facility to return a response packet to a sender with minimum overhead.

The sequence of events for transferring a packet requires the sending process to build and store the packet, call send in REX and then await the return packet until a post matching solicited ECB in REX activates a wake-up sending process. The sending REX, upon the occurrence of call send, places the packet on the bus out queue, waits for a bus output port, and then passes control to the outstack handler. Upon receipt of the return packet the sending REX activates the post matching solicited ECB. The executive services subsystem **904** polls the processor outstacks until the packet is detected. It then decodes the logical address to obtain the physical address, polls the receiving processor instack, and when available transfers the packet. The return

packet is handled in the same manner by the executive services subsystem **904**.

Upon transfer of the packet to the instack, the receiving REX executes an instack interrupt handler and posts an unsolicited ECB. The return packet is output in a manner similar to the sending REX procedure. The receiving process checks the ECB's and upon finding the packet, generates and sends the return packet.

A sending process can also define an event that informs it as to when the packet actually leaves the host processor which could be some period after it requested the packet transfer due to a queue to use the executive bus ports **517, 519**.

A method of sending a packet directly to a corresident process is provided by REX. This avoids having to queue for use of the I/O ports, and is obviously much quicker. However, no system-level check of this packet transfer is possible as the executive services subsystem **904** is not involved. Also, in the system **100** processes can "float" from processor to processor as system configuration changes, and general use of this method is not encouraged.

The first three words of the 16-word data structure of a standard packet contain the routing information needed to identify the destination and the sending process. The two 20-bit process identification codes are packed into three words, each field having the following meaning:

Word 0	FCODE	Bits 15–12
	CXID	Bits 11–8
	LBID	Bits 7–0
Word 1	ASPID	Bits 15–8
	SPID	Bits 7–0
Word 2	RFCODE	Bits 15–12
	RCXID	Bits 11–8
	RLBID	Bits 7–0

1. Process Identification of Destination—This consists of two fiels in Word **0** and one field in Word **1**. Word **0** contains:

LBID—The logical address of the destination processor (range 64–128) or the fixed address of a major system jobs' primary node, in which case this field is termed the System Bus ID. A single physical procssor can host as many System Bus ID's as it hosts system processes. System Bus ID's are in the range 2–63. Inter-Process Communication replaces either LBID or System Bus ID with the Physical Bus ID (PBID) after consulting its polling tables prior to completing a packet transfer. PBID range equals 0–31.

CXID—Context ID of destination process (0–31). Word **1** contains the Subprocess ID (SPID) of the destination process within the main context (equals zero if destination is the context main process).

2. Process Identification of Sending Process—This is contained in the upper byte of word **1** and word **2**. Note that to "turn" a packet header around requires only swapping words **0** and **2**, and swapping the bytes of word **1**.

The four-bit function codes (FCODE) relate to the sending and receiving (RFCODE) processes' function table and provides a first-level filter for up to 16 classes of packets that a process might receive.

Word **3** usually contains the Job Number of the sending process, and enables the receiving process to validate the packet's source. Word **4** can contain further fields identifying the exact identity of the packet, and Word **5** is often used for control, status and error flags. Note that, apart from the header, the definition of the meaning of the fields in a packet is the responsibility of the sending and receiving processes.

Large amounts of data need to be moved through the system when input/output operations occur. For example, if the system **100** were being used to record voice, voice data is moved from the RECORD process in a real-time processor **410, 412** along the inter-processor executive bus **912** o4 **914** and to the specific ACCESS subprocess handling the transfer in a file services processor **934, 936**. Packets are still used to effect the transfer, but these packets are built, sent and controlled by the REX's of the two processors involved; these functions are invoked by calling REX input/output service routines (IOSRs). Once REX has set up a multiple-packet data transaction, it can use more efficient packet formats than the standard format, reducing the 3-word header to two words for the majority of packets. Further, REX includes a sequence number in each packet so that the order of data can always be preserved. Packets used in mass data transfer are referred to as immediate packets as they are addressed to REX's process (i.e., Context **1**, Subprocess ID **0**). Other processes can make use of immediate packet formats to implement their own data buffer transfers, via REX's transaction management functions.

D. System Operation and Control

The time between power initially being applied to a system **100** and it reaching operational readiness is termed the system wake-up period. During this period, the system processes required to realize the particular functional configuration of the system **100** are initiated in suitable host processors. System wake-up is an automatic sequence, guided by the job scheduling and allocation monitor (JSAM), using the contents of the system requirements file (SRF) as a guide to the software components required.

Wake-up takes place at two distinct levels. First, each individual processor sequences through its own wake-up cycle moving from physical initialization through to diagnostic testing to logical initialization. This occurs whenever power is applied to a processor, or when a processor is attempting to recover from a trapped condition caused by a software or hardware error. When the processor has successfully reached the operational status, it is available for JSAM to allocate processes to it via create process requests.

The second wake-up sequence occurs at the system-wide level. This is controlled by the primary JSAM, resident in one of the two executive services processors **916, 918**. The JSAM wake-up process itself is bootstrapped into existence as one of the functions of the physical initialization program which is ROM resident in the bus control extension unit **922** of the primary executive services processor **916, 918**. Which one of the two executive services processors **916, 918** hosts prime JSAM is determined by which one completes physical initialization first.

The sequence of events comprising system wakeup from this point onward are as follows:

1. JSAM allows five seconds for other processors in the system to initialize and report their operational status to JSAM. JSAM then assigns logical bus ID's to each live processor.

2. JSAM selects a File Services Processor **934, 936** (FSP) that has attached to it a disk drive **930, 932** with a system volume mounted (this is reported to JSAM as

part of the file. services processor's wake-up status message). The selected file services processor **934, 938** is instructed by JSAM to continue initialization by loading and running the System Directory Manager (SYSDIR). SYSDIR, as part of its wake-up sequence, initializes the other related processrs in the file services processor **941, 946** of the file services subsystem **908** (a possible maximum of **26**).

3. JSAM requests from SYSDIR a copy of the system requirements file (SRF).

The system requirements file fonctains a list of the processrs required to form an operational system. JSAM inspects its list of operational processors and sends create process requests to each processor that provides a resource sufficient to host an entry in the system requirements file. Major system components are started first, and system bus ID's are assigned to them. JSAM sets itself a time-out for this operation and any unsatisfied system requirements file entries at the end of this period are reported to the operator console for consideration. This can occur if a necessary processor fails to wake up.

The system requirements file is maintained by JSAM after wake-up during the normal operation of the system. If a new processor reports to JSAM at any time, the system requirements file is inspected to see if any outstanding processes need the resources provided by the new processor. For example, if the processor hosting the back-up copy of JSAM fails, the absence of a back-up JSAM is noted in the system requirement file. When a replaced of repaired processor capable of hosting JSAM or executive services processor **916, 918** reports itself operational, JSAM will automatically create a back-up JSAM in it.

Due to the very high availability of the system **100**, wake-up occurs at very irregular, infrequent intervals. During normal system operation, re-configuration is accomplished by editing records in the requirement and configuration files from a system terminal **270**, JSAM can then dynamically adjust the system to reflect the new configuration.

Specifically, the Systems Programmer can:

(1) Create and send a packet to any process.
(2) Display scratchpad, program and data memory.
(3) Alter scratchpad, program and data memory.
(4) Load a program into a processor.
(5) Jump a processor to a program memory location.

Human control and interaction with a system **100** is via system terminals, such as system terminal **270**. A number of different types of system terminal are supported, each providing varying degrees of access to the system. Physically, each terminal consists of an intelligent visual display unit (vdu) and keyboard with a multipartitioned screen that allows for a number of simultaneous display/interaction areas. After providing a log-on iser ID and password, a system log-on menu is displayed. The choice selection of the user is checked against his usage rights. Providing ID, password and usage request agree, a system terminal **270** with a selected level of access and control is presented. Four types of system terminals are supported:

1. Systems Programmers Terminal—The system programmer's terminal gives the user full access to all of the functions and utilities available at a system operator's terminal. In addition to the normal operations-oriented functions, a System Programmer's Terminal can be used to interact directly with any of the Delta's processors, providing a software front-panel control.

2. System Operator's Terminal—Table 1 shows the function and utilities available via this terminal. The System Operator's Terminal is used for day-to-day control of the Delta by the System Manager. The functions include:

(1) Set Time-of-Day, Date
(2) Remote Card Reader Interface
(3) Program Librarian Interface
(4) Program Loader Interface
(5) Change Logpac Parameters
(6) Change Alarm Message Parameters
(7) Remote Data Structure Access
(8) Disk Device Manager Operator Communications (DVOLCOM)
(9) Magnetic Tape Device Manager Operator Communication (MTVOLCOM)
(10) System Spool Manager Operator Communications (SPOOLCOM)
(11) Program Library Maintenance
(12) System Log Manager Operator Communications (SLOGCOM)
(13) Add a Device, Remove a Device (Smart/dumb terminals, printers, real-time lines, SCS channels, etc.)
(14) Mark a Device Inactive

3. Operator Terminal—Provides a sub-set of the capabilities of the system operator's terminal, together with functions for maintenance, field service, updating user ID's and passwords, etc.

4. System File Maintenance Terminal—Provides the capability to update records in system files via an interactive dialogue at the terminal. Each record update request from the terminal is first verified with the chief system user of the file before acting upon the request. System files maintained from this terminal include:

(1) System Configuration File
(2) System Requirements File
(3) System Device File
(4) Program Library Descriptor File
(5) Function Library
(6) SYSDIR Descriptor File
(7) Node Descriptor File
(8) System File Maintenance Terminals' own descriptor files

The system file maintenance (SFM) terminal can be easily expanded to allow maintenance of other system and application files. It is table driven and can be used to update its own descriptor files. These descriptors not only specify the record types to be updated, but also the associated interactive screen display and dialogue.

As previously mentioned, logically different terminals use the same physical device—an intelligent vdu/-keyboard with a multi-partitioned screen. The partitioning and function of each screen area is standard for all terminal taypes as is the man-machine dialogue. The various screen areas may include utility name and status, system status, a scrolled system log display. utility partition no. **1**, utility partition no. **2**, function prompts, a menu and error messages.

Simple functions can be invoked from the keyboard by pressing one of five function keys, which activate one of five options presented in the Function Print-/Menu partition. The functions in the menu can be switched between a large set of groups of five functions.

More complex utilities interact with the user via two dedicated partitions in an individual fashion, but each utility uses the same command interpreter accepting from the keyboard commands with the general form:

COMMAND P1, P2, . . . PN;Q1,Q2, . . . QM
Where
COMMAND is a utility-specific command
P1 . . . PN is a parameter passed to the utility
Q1 . . . QM is a parameter passed to the command interpreter.

One screen partition is used to display log packets being sent to the system log (SYSLOG). As SYSMON is the backup process to SYSLOG, these packets are available for immediate display at system terminals. As log packets are generated for specific events by all system processes, the system terminal **270** can show an associated message describing the event that caused the generation of the packet. The screen partition can be scrolled through a buffer of received log packets. A permanent copy of log packets is obtained by printing the contents of the system log via a utility involved from a system operator's terminal.

The System Terminal software can be used to develop application-specific utilities that can be invoked and used from a system terminal. Specifically, the following services may be implemented:

1. Invoke a user-supplied utility (i.e. a process) by name.

2. Provide ID and password security check against a user supplied criteria.

3. Allow use of the utility screen partitions as either two 10×40, one 10×80 partitions or one 2×80 partition.

4. Allow the whole screen area to be used, apart from the status and error message partitions, for the utility.

5. Provide a display buffer for any partitions previously defined above, and automatically maintain the screen partition to reflect the buffer's content.

6. Provide a command interpreter interface between the utility and the terminal.

User-defined utilities provide a convenient method of controlling and interacting with application systems, while maintaining a consistent human interface to all control functions.

The collection of processes providing system terminal functions are collectively referred to as the system monitor (SYSMON), and are an example of a process network. Each terminal manager invoked to interact with a specific physical terminal is implemented as a separate subprocess of a primary node process and a number of co-resident subprocesses provide common functions. The current version of SYSMON is source-written in Pascal and resides in a number of Ideal Machines in a general purpose processor (One IM per process). SYSMON shares a system bus ID pair with SYSLOG. SYSLOG is the primary system bus ID of the pair and SYSMON the secondary system bus ID.

If either SYSLOG or SYSMON fail, log packets are re-directed to the survivor which stores the packets until the failed member is restored, and the stored packets are transmitted to the new copy.

The program development system generates SPM or P-Code files as a result of compiling/assembling/linking operations. These code files are loaded by REX into processors in response to create process requests. A mechanism which links the products of the program development system and the requirements of REX, also allows human intervention at the system level for operational and management purposes. Function and program libraries consist of indexed string datasets resident on system or user disk volumes.

It is possible that several different variations of a program may exist, all performing the same function but each with different characteristics. This is especially true in the system **100** which has a number of different processors each with unique hardware attributes. For example, the system error logger as a function could have two programs available. One realized in SPM machine code, and capable of running in any processor; and a second, identical function program, written in Pascal and realized in P-code, only capable of running in a general purpose processor. Both of these programs would, however, be indirectly identified by the same function ID.

When a user requests a node creation as part of setting up a process network, a function ID can be specified rather than a specific program ID to realize the node's process. JSAM can consult the function library, using the function ID as a keyword. The entry corresponding to the function ID keyword consists of a list of program descriptors each give a detailed description of a program which could fulfill the requirements of the function. JSAM thus has a set of programs that it can match against the known resources of the system **100** when attempting to satisfy a create node request. After performing a dialogue with the function library, JSAM uses the selected program ID in its request to the chosen host processor to create a process.

A function ID consists of three fields:

1. A Function reference number in the range 0–64.

2. A System Version in the range 0–255, used to select between a number of Released Versions of the same program.

3. A Mode value, range 0–255.

User processes can interact directly with the function library to scan potential programs. Furthermore, if a program ID for a specific processor-type is required, a function library name can be used which specifies the processor type (via its normal extension board). The specific program description entry for that processor is returned. The user can specifically request a particular program ID in the create node request to JSAM in which case the function library dialogue is bypassed. The function library is resident on the system volume (duplicated, of course, on the system backup volume). The contents of entries in the library can be manually maintained from a system operator's console, and a degree of automatic maintenance occurs when changes are made to the program libraries (i.e., if a new version of an existing program is created and entered in a program library, the function library is conditionally updated to reflect the change).

The function library provides a convenient logical separation between functions and programs which can be a very useful feature in the development phase of a project, or to provide better capability to survive failure by providing options to get around specific processor-type nonavailability. Note that a number of function IDs can be mapped to the same program, a useful feature in test environment.

The interface to the function library is provided by the system directory manager (SYSDIR).

When source code is assembled and linked together, the output of this process is referred to as a load module. A load module contains from one to 16 separately relocatable pieces called relocatable modules. Each module is limited to 4096 words. The number of relocatable modules is further limited by size constraints in the load request.

A load module may be altered by changes called patches, which may later be removed from the load module. Each load module and related patch set constitutes a unique program on the system and therefore constitutes a different historical version of that program.

Since programs interact with other programs on the system and interfaces sometimes change, there is also a need to coordinate different versions of different programs. Therefore a system version refers to this coordinated change.

All Programs reside in data sets called program libraries. There may be several different program libraries in the system to allow distinctions such as system version, application type or ownership.

A program library is an indexed data set contained on either system or user volumes. The data set consists of three different types of records. There exists one program header record for each program in the library. This record contains program related information, which includes the latest load module number and latest historical version number for the program.

Each historical version of a program has a corresponding program version header. This record contains information relating to that program version including the names of the pieces that compose that version (i.e., load module and patches) and possibly a preprocessed version of the program called a load program. This processed copy of the program has all the patches (if any) applied and is in a format that allows efficient relocation and transfer of the program. Only those versions that are frequently called have load programs built and retained.

The third type of record is a load module record, which contains a particular load module and all the patches that have ever been applied to that load module.

The records are indexed by a key which contains the record type, the program number, and the NEB type and version. Program version headers also have the historical version number in the key. Load module records contain the load module number in the key.

A particular historical version of a program can be uniquely identified in the program library by its program library name which consists of the program library ID, the program number, the NEB type and version and the historical version number.

Programs are also identified by their external ID. This ID is used as an external interface for human interaction. This ID consists of a part number, version number, date program loaded and an alphanumeric nickname.

In order for a processor to execute a process, the program code of that process must exist in program memory. When it is not in memory, the program is loaded by a request to SYSDIR. This request may be sent in response to a create process request by the REX of the processor that wants the program loaded, or it may be forced by a process in another processor.

The requestor passes to REX the program ID of the program to be loaded. This ID might either be known by the process making the request or have been passed to it by JSAM which learned the ID from a function information request dialog, as explained earlier. If the Program ID is a function library name, SYSDIR is used to obtain the program library name by reading the function library record.

There are a number of different types of load requests. A normal load is one where the requestor (i.e., REX) first requests the program load information (i.e., memory and resource requirements) and is then returned the memory addresses for loading the program. A forced load allows the requestor to make a single request to load the program, specifying the load addresses in the request.

Within the scope of normal and forced loads, a primary load is when a program is being loaded to a processor and no other copies of that program code are currently loaded in the same processor. A secondary load (for those programs that allow it) is a load related to a program that already has program code loaded. This will load only process-related data memory modules, which are needed to handle multiple invocations of the same program. Finally, an overlay request is a request to load selected code modules. This mechanism is used to allow the partitioning of a program such that all its code segments do not have to be resident in program memory at the same time. An overlay request is a request to replace one or more of these overlayable pieces.

The 16 relocatable modules in the load module of the program may be of four different types. These are:

(1) Primary program memory modules
(2) Overlay program memory modules
(3) Program-related data memory modules
(4) Process-related data memory modules

The different load types outlined above cause different combinations of these modules to be loaded into memory. In a normal primary load, primary program memory, program-related data memory and process-related data memory modules are loaded. A secondary load is a request to load only process-related data memory modules, as the other modules containing program code will be shared from a previous primary load. An overlay load is a request to load some of the overlay program memory modules, as indicated by an accompanying overlay map.

Process-related data memory modules allow an invoked process to access loaded and initialized private data. This saves a process from having to acquire the memory space dynamically and then initialize it. The initialization can be done at assembly time and the memory acquisition effectively accomplished via a program load.

Process-related data memory can also be used to hold preinitialized program control information which will be used by the system (e.g., process control block, event control blocks).

It is the requestor's responsibility to manage and control secondary and overlay requests. That is, the requestor must know that a primary load was performed, that the modules are in memory and where they reside so that any subsequent relocations can be performed on the new pieces. For overlay calls it is the requestor's responsibility to know which pieces can overlay which pieces, to keep track of which pieces are currently in memory, and to mechanize transfers of control between separate overlayable pieces. However, in normal system operation, all program loading is managed on the users behalf by REX, and this is only of concern in forced load requests which bypass the REX of the target processor.

INPUT/OUTPUT SERVICES

A. Overview

The system **100** is a multi-media computer system which utilizes failsafe architecture to provide very high levels of availability and uninterrupted processing. It is a tightly-coupled, distributed network of multiple high speed processors, interconnected by a high speed packet switching network, and a fully distributed fault tolerant operating system that together provide a uniquely flexible, high throughput processing system. The modular architecture of the system allows the system to be configurated to accommodate the most demanding input/output and processing requirements.

The unique hardware/software architecture of the system **100** enables it to operate in environments which mix real-time, voice communications, data communications, computational, interactive and transaction processing. This fully integrated hardware/software design eliminates the need to interface uniprocessors, store and forward nodes, etc. to obtain the desired system capabilities.

The function of the input/output services of the system **100** is to manage all information transfers between processes and attached devices. This includes the interconnection of real-time voice or data channels, acquisitions of real-time data streams, data storage transfers, and control of and communication with large numbers of external devices. To accomplish this the system **100** utilizes multiple microprocessors, microcoded in ROM, to control the internal high-speed packet switching as well as the buses from the real-time, interactive and file services subsystems. Further a comprehensive set of I/O services routines (ISORs) is available to processes running in an ideal machine (IM). These can be used to acquire, control and communicate with any device attached to the system **100**. The operational protocols associate with the ideal machine I/O service routines are described in greater detail in Section B. A process, once it obtains control of an external device, has a direct link with that device.

Applications, running in an ideal machine use the I/O services of the ideal machine monitor (IMM) which directly use the host processor's resident executive (REX) I/O service routines. Privileged users, working outside the IM environment creating assembler programs, can call directly on REX I/O service routines. It is at this level that a system designer would code customized device handlers.

Two kernel system software components are involved in I/O transactions. Before a user process can communicate with a device, ownership of the device must be acquired by the user process, or by some other process within the same job network (i.e., a process with the same job number). A user cannot talk to or control devices not specifically owned by a job of which the user process is a member. Device ownership is achieved by a request to the system device manager, (SYSDEV). SYSDEV checks to see if the device is available, and if so, sets up a logical path between the requestor and the device handler of the required device. Transactions can then take place between the user and the device handler directly with no intervention by SYSDEV until the user returns the device to the "pool' of unowned devices.

The second kernel system process involved in I/O transaction initialization is the System Directory Manager (SYSDIR). When a user wishes to transfer data to or from a device channel or dataset, the user must first issue an Open request. In the case of datasets, the open request is routed to SYSDIR. SYSDIR checks that the requester has access rights to the named dataset, and on which volume (disk or magnetic tape) it exists. SYSDIR builds a communications path between the user and the logical I/O handler of the dataset through which all subsequent transactions are handled.

The device and dataset handlers reside in the processors to which the device channels or data volumes are physically attached. The handler receives the packets generated by the REX of a processor requesting input/output services and provides the logical to physical interface functions needed to implement the request.

At any one time, the system **100** can be hosting a large number of independent jobs with each job acquiring and opening any number (within system limits) of devices and datasets. Device handlers can accept concurrent requests for devices, and each handler can manage multiple devices of the same class. Individual datasets can be opened for simultaneous access by multiple users and a range of access restrictions can be specified. Spooling is also available for queueing output to a specific device type such as a printer.

B. Input/Output Protocol

A well-defined protocol exists for using devices and datasets either through IMM Services or directly through REX IOSR's. Privileged users (those operating outside of the Ideal Machine environment) perform I/O functions directly using REX IOSR's, while other users (those operating within the Ideal Machine environment) use the services of the ideal machine monotor (IMM) which in turn calls REX IOSR sequences to perform a particular function. These IMM IOSR functions exist as groups of intrinsic Pascal procedures, each group addressing a particular area of the I/O protocol and are listed as follows:

Acquire Device Procedures:
VACQNAME: Acquire by name.
VACQLIST: Acquire by type and sub-type.
Data Set Maintenance Procedures:
VCREATEDS: Create a new dataset.
VMODIFYDS: Modify an existing dataset.
Open Procedures:
VOPEN: Open a device for I/O.
VOPENDSET: Open a dataset for I/O
Read Access Procedures:
VREADDEV: Prepare a device for a Get.
VREADCRT: Capture a CRT screen.
VREADDIRECT: Input from a direct dataset.
VREADKEY: Input from indexed dataset.
VREADNEXT: Read next record (indexed set).
VREADPREV: Read previous record (indexed).
Write Access Procedures:
VWRITEDEV: Prepare a device for a Put.
VWRITEDIRECT: Output to a direct dataset.
VWRITEKEY: Output to an indexed dataset.
Data Transfer Procedures:
VGET: Transfer from device or dataset.
VPUT: Transfer to device or dataset.
VTRANSFER: Transfer between device or dataset.
Device Control Procedures:
VCONTROL: Send control and receive status.
Data Set Control Procedures:
VDELETEKEY: Delete a keyed record.
VRENAMEKEY: Rename a keyed record.
VALLOCATE: Mark direct records in use.
VRLSELEM: Release direct records for use.
VADDEXTENT: Increase extent of a dataset.
VMOVEWINDOW: Move window in subfile.
VINSERTMARK: Mark a record in a subfile.

VDELETEMARK: Remove a mark in a subfile.
VINSERTELEM: Create a new element in subfile.
VDELETELEM: Delete an element in a subfile.
VMOVELEM: Move a set of subfile elements.
Access Completion Procedures:
VENDIO: Terminate a data transfer.
Close Procedures:
VCLOSE: Close an Open device or dataset.
Device Release Procedures:
VRLSEDEVICE: Release ownership to system.

For privileged users, REX provides several calls for both device and dataset manipulation. The way in which one of these basic IOSR calls is used is determined by the parameters supplied in the call. REXs' basic set of IOSR's include the following functions:

(1) OPEN
(2) CONTROL
(3) ACCESS
(4) GET
(5) PUT
(6) TRANSFER
(7) ENDIO
(8) CLOSE
(9) GETIOSTATUS

The above list is not exhaustive and REX provides functions which combine sequences of the basic set for convenient use in commonly occurring I/O operations. In general, each call to a REX function can choose between a number of variations of the functions basic service.

There are two types of calls to REX. One suspends the calling process until the request is complete. The second returns control to the caller immediately after the request is initiated and provides the program origin at which processing will recommence once the request is complete.

This does not apply to I/O calls from applications jobs running in an ideal machine where a process always suspends until the request has been completed. This avoids concurrent operations building up in a single ideal machine. Concurrency in the system **100** is obtained by running concurrent singular processes each in a separate ideal machine.

When a level **3** application process performs I/O, it communicates with a level **1** process which provides a logical interface to a device or dataset. However, before this communication can begin, access to the handler has to be gained by way of the Kernel System. Devices have to be acquired from the system device manager (SYSDEV) and datasets have to be created and opened by way of the system directory (SYSDIR) catalogue.

To build a logical control channel to a specific device (excluding datasets but including telephone lines in voice applications), the device must first be acquired by the process wishing to send or receive control messages. There are two basic ways to acquire a device.

Acquire-by-Name—Each physical device is uniquely identified by its termination point at the system **100** interface. If more than one channel to a device exists (as occurs, for instance, with operator stations and remote line concentrators), any of the devices' channel identifiers can be used to name the device. Acquire-by-name is used when a process has been specifically invoked as a result of the system detecting activity on an unassigned device. The channel ID on which the stimulus was received is passed to the invoked process as part of its start-up parameters. The process acquires the device by name and communicates with its device handler to effect whatever subsequent action is predefined for the process.

Acquire-by-Type—A process may need to acquire a device with certain physical or logical characteristics without regard to which particular device is assigned to provide them. For example, if a magnetic tape drive is needed, the user is not particularly concerned with which drive is assigned, as long as one is made available to receive data. In these circumstances, the user supplies a device type code that identifies both the physical device type required—(e.g., terminal, tape drive) and also a hierarchical list of sub-type groups, each of which identified one or more equally acceptable logical subtypes within the physical type (operator's terminal, supervisor's terminal, etc.)

The acquire request, together with the device channel ID or the device type code and sub-type list, is sent to SYSDEV. If a suitable device can be found, the device channel handler is informed and SYSDEV passes back to the user the process ID of the handler, the channel identified for each channel plus tupe and sub-type codes of the device. If the requested device exists, but is already allocated, the user has two options (indicated in the acquire request):

(1) To be informed the request was unsuccessful with no further action.

(2) To have the request queued (with a queue priority and time limit supplied by the user) until a suitable or specific device becomes available.

If a requested device does not exist because it is out of service or becomes so after a request has been queued, the user is notified that the request cannot be fulfilled.

The Create function allows a new dataset to be formed with a number of attributes defined by the user in the create request. The modify function allows the attributes of an existing dataset to be subsequently changed. A create request is passed to the system directory manager (SYSDIR) which maintains a catalogue of all datasets known to the system **100** either on disk volumes or magnetic tapes. After validating the request, SYSDIR enters the dataset details in the system directory and passes back to the user an indication as to the success or failure of the request. Failure to create a dataset could occur if a dataset with an identical name already exists, if insufficient space exists to satisfy the initial space needs of the dataset, or if illegal attribute values were supplied in the create request.

The attributes that can accompany the create request include:

(1) Dataset name
(2) Dataset type
(3) Hardware type (disk, magnetic tape)
(4) Creator's identity (often log-on user code)
(5) Access security information (covering groups of potential users)
(6) Action regarding duplicated datasets
(7) Create or add to existing dataset
(8) Volume ID (if Add)
(9) Allowable extents
(10) Directory characteristics
(11) Record parameters
(12) Initial size required (if Create)

The Modify request enables the above attributes in an existing dataset directory entry to be changed by the owner.

A user can request exclusive use of a complete disk volume (magnetic tapes have single user access by nature). If a user requests creation of a dataset without

specifying a "Pack ID" of a volume owned by the user, the dataset is created on any suitable public volume.

Devices no longer required by a job can be made available for use by other users with the release function. Any subsequent attempts to communicate with the device using the reference data allocated by the original acquire function will end in rejection.

If a job terminates without releasing previously acquired devices, they are automatically released. This is initiated by the job scheduling, allocation and monitoring kernel function (JSAM) which always informs SYSDEV when a job has terminated. SYSDEV in turn informs the device handlers to purge their tables of ownership of the terminated job. However, the presence of unreleased devices is an indication of a potential fault condition, so such occurrences are logged for analysis.

If a job terminates with open datasets, these are closed by the system directory manager (via JSAM request).

Once the initial dialogue with the kernel system is complete, the user process communicates with a level **1** device or dataset handler which is resident in the processor to which the device channel or dataset volume is physically attached. When a dataset is initially opened, the open request is first passed to the kernel system process SYSDIR before a level **1** handler is invoked. The following functions apply to owned devices or pre-existing datasets to which the user has the required access right for the specific request.

An open request establishes a communications path between the user process and the device channel handler or dataset access process. In the case of datasets, the user supplies his user ID along with the dataset name. These are checked against the descriptions in the system catalogue to verify the validity of the open request. In the case of devices, the user must supply a job number and the appropriate channel ID of the one or more obtained when the device was acquired. If the device channel has already been opened by a previous request and not closed, the current open request is rejected.

The open request establishes contact between control processes at each end of the communication channel which handle the subsequent data transfers.

Once an open has been performed, multiple read and write accesses may be performed by the opener on the device or dataset.

Read, write and update access requests establish the conditions prerequisite to the transfer of data. They check on the physical availability of the device or dataset for date transfer (it could be busy with a previous transfer) and effect all of the required actions at both ends of the channel necessary to permit a subsequent data transfer. This can involve the creation and loading of buffers at the data source. For example, in the case of dataset I/O, a read access request would specify a logical record in a file. As a result, the required data is transferred from the disk into a buffer in the file services processor **934**, **936** (FSP). Any subsequent "GET" functions effect transfer of the data from the buffer to the requestor. A write access request to a dataset allocates an empty buffer into which a logical record will be placed. An update acts initially as a read, but the buffer which holds the user data is retained to permit rewriting of all or any portion of the data back to the disk. In other cases, external buffering is used. For example, with the magnetic tape, the buffering is provided in the magnetic tape controller. Note that for operator stations, data buffering is assumed to exist in the terminal associated with the station whereas no audio buffering is assumed for the station headset.

Put and get request functions request the physical transfer of data between the user process and buffers in the device, channel controller, device handler or dataset handler that were previously primed with a read or write access request.

With a transfer request, a user process can initiate a transfer of data between a source and destination channel or dataset without passing the data through the user's process. Thus a user process can effect a data transfer, for example, from a display record on disk to an operator station CRT with no intervention required by the user once initiated and with direct routing of data through the system **100**.

An access completion request clears the access path, terminates the access and returns to the system any associated buffers. To validate the request, particularly for multiply-accessed datasets, the user specifies the access reference value obtained from the system when the read/write access was requested. When a sequence of accesses is needed, each access completion request can also be used to initiate the next access.

A close request deallocates the access control capacity which was assigned as a result of the initial open request from the user. Close clears the communication path between the two. To reaccess the device or dataset (except for device control only) the user would have to reinvoke the open function.

Control data can be sent to devices and datasets by way of IOSR control functions. This information may be trapped by the device or dataset handler, or may be transferred directly to the device itself.

The minimum requirement to be able to control a device is for the device to be acquired, whereas a dataset also has to be opened before any user interaction can begin. Acquiring a device provides the user with enough identification and authorization to talk directly to the devices handler. This is especially useful in managing devices that do not generate or transfer data. For example, the circuit switching function of the real-time subsystem **230** does not pass data through the system **100**. It interconnects real-time data channels. A user who has acquired a set of these channels can send control parameters to the circuit-switcher specifying the interconnection required without having to prepare buffers or initiate control processes with Open/Close, Read/Write requests.

IMM IOSR's provide a group of control functions that simplify the control of the more complex dataset types available on the system **100**. These functions provide access to the record-control sections of datasets, allowing modifications to be made to the structure of a dataset without having to read records from the dataset, reducing the amount of physical disk I/O required.

For privileged users working outside of the ideal machine environment, REX provides a single parameter driven procedure for all control functions.

An acquire request returns the ID of a devices' channels and also the process ID of the devices to the user handler. A privileged user can then use basic packet transfer services to communicate with a device or dataset handler in a totally self-structured way. This provides greater flexibility but transfers to the user the responsibility for the correct packet-level protocol be-

tween the user and the device handler which is normally the responsibility of REX.

C. Devices

In general, a singular device can include more than one channel attached to the system **100**. For example in a voice application, an operator station consists of a duplex data channel to a CRT and associated keyboard plus an audio channel to and from the operators headset. If an application job needs an operator terminal, it has to acquire the two channels simultaneously since it makes little sense to attempt to utilize the two independently. When a complex multi-channel device is requested, SYSDEV's device configuration tables list the set of attached physical channels. If any one of the channels is unavailable, another device is selected if an acquire-by-type is being performed, or the acquire request is either enqueued or rejected if an acquire-by-name is specified.

If the device is successfully acquired, SYSDEV returns to the user the channel ID and process ID of each channel associated with the device. This effectively defines the communications link from the user to each of the devices channels. The return link is established in the open request and each control request.

The interactive service subsystem **908** can support:

(1) Smart terminals
(2) Dumb terminals
(3) Line printers
(4) Magnetic tape drives
(5) Synchronous and asynchronous channels
(6) Line concentrator data links

It should be noted that the magnetic tape unit is a device that supports datasets. Before opening a tape dataset, the tape drive is first acquired by the system directory manager (SYSDIR) on the user's behalf. A dataset open failure response will be returned to the user if SYSDIR is unable to acquire a drive. Operator requests to mount off-line tape volumes are passed to the system monitor by SYSDIR as part of the open process. Tape datasets conform to ANSI X3.27 format, allowing archived data to be transported to other processing systems.

A number of printers can be attached to a system **100**, of which a minimum of two are usually assigned as system printers. These are not normally acquired directly by users, and are owned by the system spooler. Data for printing on a system printer is passed by the user to the spooler with a print request header, with the spooler managing the transfer of print data to the printer. Non-system printers can be acquired for private use in the same way as any other assignable device. The printer handler can operate in three modes.

Transparent mode—with all printer control characters embedded in the data stream.

Edited transmission mode—in which the user sends data in message blocks with the printer handler formatting the data according to a layout defined previously by the user.

Line-oriented transmission mode—where data is sent on a line-by-line basis, each line containing format information in the first word.

The Printronix printer normally used with the system **100** contains a vertical forms unit that can be loaded with predefined form layouts. A graph plotting mode can be entered from the transparent mode.

The Real-Time Subsystem allows real-time data channels to be attached to the system **100**. Some of the functions provided to the user include:

(1) circuit switching,
(2) data recording,
(3) data playback,
(4) signal processing, and
(5) supervisory signallying and control.

One use of the real-time channels is for PCM-encoded voice with a transfer rate of **64** K-bps. Equally, any other information that uses the normal analog bandwidth of the telephone system could be switched or processed by the same functions. Variations of the real-time channel interfaces enable non-PCM encoded data to use the sybsystem—(e.g., telemetry, video, etc.) The bandwidth available to a channel can be varied at the expense of the total number of channels that can be switched or processed.

Users can acquire real-time channels either by type or by name. Individual circuit switching and control can be accomplished with a simple acquire-control-release protocol. Supervisory information and control data can be received from and sent to owned channels. In telephony applications this would be used to set dial tone on a line, for the detection and generation of dial sequences, setting on-hook and off-hook conditions, for seizing lines, etc. Furthermore, a number of channels can be logically grouped by the owner and referred to by a network ID. Within a network, one-to-one, one-to-many and many-to-many interconnections can be set up by the user simply by providing interconnect maps to the circuit handler. Thus, in voice applications cross-connects, broadcasts and conferencing can be implemented through control functions. A complete network interconnection can be torn down with a single command by referencing the network ID.

Signal processing functions such as record and playback are invoked as transient processes in the real-time processors **410, 412** (RTPs). User-written signal processing functions can similarly be installed in the real-time processors **410, 412**, which has access to the data channels attached to the real-time subsystem **230**. Each real-time processor **410, 412** can handle 16 input and 16 output channels. Typical real-time processor applications could include spectral analysis, filtering, and the assembly or reassembly of subcommutated channels.

Record and playback functions allow data from real-time channels to be stored on disk and subsequently reconstructed as a real-time signal. To minimize disk storage requirements, voice messages and conversations can be compressed during record, and decompressed during playback by a real-time processor **410, 412** with a compression ratio of better than **2** to **1**. It would also be possible to retrieve the data from disk for manipulation and processing by an independent job. During record, record markers are inserted in the data stream at approximately one second intervals, allowing the stored data to be the subject of a record orientated processing operation (e.g., editing stored voice messages). The record and playback processes function much as if they were simple mechanical recording devices, although in practice they are implemented as software processes hosted by real-time processors **410, 412** (RTP).

Each real-time processor **410, 412** can concurrently support approximately 8 record processes and 16 replay processes. The job scheduling, allocation and monitoring kernel function (JSAM) causes these processes to be initiated in response to a request for a record or playback function. The owner supplies a dataset name and channel ID for the transaction, and control information to operate the machine—initialize, record, replay, pause

or terminate. The software "recording machine" interfaces to both the real-time channel handler and the dataset handler to actually implement the transfer using standard REX IOSR's. In addition, the channel that is the object of the record/playback transaction can, at the same time, be involved in a circuit-switch network.

D. Datasets

Datasets recorded on magnetic tape volumes conform to the ANSI X3.27 standard, initially with level **1** support on single or multiple tape volumes. Tape datasets are restricted to direct access, fixed record length. Each tape has an ANSI standard label which includes a volume ID.

Disk datasets comprise an integral number of physical disk tracks. When a dataset is created the expected number of tracks required by the user is specified in the create request. This initial space allocation (or extent) is in the form of contiguous tracks on the volume, making accesses to the dataset optimized for high speed. Access speed and transfer rate are factors which dominate over space allocation efficiency, especially in communications environments. Further extents for a datasets' records are requested if more space is needed.

As an extent is acquired, it is soft formatted into a series of blocks, the block size being a dataset attribute determined by the user. Since the disk drive controller transfers data in units of blocks, the block size is set to trade-off access time versus buffer requirements in the file services processor **934, 936.**

A further parameter supplied by the creator is the record element size. This is the unit of logical allocation of disk space within a block and the block size must be an integral number of record elements. The various logical record types are all built from allocating record elements within blocks within the dataset extent.

The system **100** supports four different types of dataset organizations, six logical data structures, and three record types. The hierarchical aspects of dataset organization are as follows:

3	DATASET ORGANIZATION:	Single Volume, singular Multiple Volume, singular Single Volume, duplicated Distributed.
2	LOGICAL DATA STRUCTURE:	Block Direct Access, Mapped Direct Access, Index Only, Indexed integral, Indexed sub-file, Indexed chain.
1	RECORD TYPE:	Fixed length, Integral records, Chained records, Subfile records.

The user can either select a combination of characteristics to build efficient structure that are specifically tailored to individual applications (such as voice store and forward), or can write a higher level of access and control which uses the basic services provided by Delta's I/O system to realize a more general data organization, such as a database. The services provided by the system directory manager (SYSDIR) and the file services processors (FSPs) is comprehensive enough to make either type of task both concise and efficient.

Dataset types can be categorized into direct access and indexed. In direct access datasets, individual records are accessed by a numerical (ordinal) address within the dataset. Indexed datasets allow access by keyword, and also allow considerable manipulation of the datasets' record structure without actually reading or writing data records to and from disk. This is made possible by the ability to access the control structures of the dataset as well as actual data records. These control structures not only include the index keys, but also logical maps of the datasets' internal record layout, and mark tables that allow locations within the dataset to be specifically "marked" for future reference. The user has full access to these control structures in the various indexed type datasets allowing considerable scope for complex record manipulation with minimum physical data movement. It is also possible for the records within an indexed dataset themselves to be direct-access files of any length allowing "access-by-name" followed by "access-by-record-number" to be accomplished within a single opened dataset. Additionally, an indexed dataset such as those used for voice recording can contain records of arbitrary length, comprising of multiple linked blocks.

The system **100** supports multiple on-line disk and tape volumes. Further, these volumes can be attached to different file services processors **934, 936** (FSPs) or interactive services subsystems **252** (ISSs). To build a communications path to a specific dataset, open requests are passed to the system directory manager (SYSDIR) for routing to the appropriate destination.

SYSDIR maintains dataset directories and manages the construction of communications paths to datasets on the various volumes. SYSDIR maintains the master volume directory and the system catalogue which contains an entry for every dataset known to the system, whether on-line or off-line, together with the one or more volume IDs on which it was created. When a request to open a dataset is received, SYSDIR checks if the volume containing the dataset is on-line, and if so, to which file service processor or interactive services subsystem it is attached. If the requestor has access rights to the requested dataset, the file service processor hosting the volume containing the dataset is requested to initiate a process to which subsequent Read/Write access requests are passed by the user. The ID of this process is returned to the requestor, completing the open process.

A volume that contains the master system catalogue is referred to as a system volume. Two such volumes normally exist and each also contains the various system program libraries and data files required to run the system **100.** The two system volumes are attached to separate file service processors **934, 936.**

Each attached volume, including system volumes, have a volume directory that contains the details of datasets resident on that specific volume. When a volume is mounted and comes on-line, its volume ID is passed to SYSDIR so that SYSDIR can decide which datasets are available for opening.

When a new dataset is created, a 6-byte user code is supplied by the creator. The owner can control subsequent accesses to the dataset and can specify that the dataset be accessible only by the owner, be publicly accessible by any user, or have limited access as specified in an associated 16-bit access rights code.

This allows subgroup access to four different capabilities: read, write, modify, and remove.

SYSDIR supports four basic dataset organizations, any of which can be selected when a dataset is created. These various options include:

(1) Single volume, singular dataset—This is a dataset that is confined to one volume, with only one copy of the dataset in the system.

(2) Multiple-volume, singular dataset—A multipart dataset that exists on a number of volumes. Only one copy of each part exists, and each part is opened individually by referencing the dataset name and volume ID.

(3) Single volume, duplicated dataset—A dataset that the system automatically duplicates by creating and maintaining two identical copies of the dataset on different volumes mounted on separate file service processors **934, 936**. Duplication is transparent to the user, and requires no special management to recover if one copy becomes unavailable for a period. The second copy will be automatically updated to reflect the latest state of the surviving dataset. (This does not involve bulk copying of the survivor. Changes that were made to the survivor while the second was off-line are marked and only those marked are transferred.)

(4) Distributed dataset—A dataset whose records can exist on a number of volumes. This type of dataset can be created to receive long records so that load-sharing of file service processors **934, 936** occurs. This would typically be required when real-time voice data is being stored. A single voice record needs continuous data storage for extended periods of time. With a distributed dataset, the system selects, for each individual Open of the dataset, a volume and associate file services processor **934, 936** that is currently least loaded. The record is then allocated to that volume. The user process must preserve in an application-maintained directory the particular volume to which the record was sent (this information is returned in response to the open request).

After a number of accesses, the individual records of a distributed dataset become distributed across multiple volumes independent of accession order. The information that exists in the directory for this type of dataset contains a list of volume IDs that contain records for the dataset.

To maintain a duplicated distributed dataset, the user may also request a concurrent Open on the next "least-busy" drive controlled by a different file service processor **934, 936**. In the case of distributed datasets, however, individual record duplication is managed entirely by the user. SYSDIR simply selects the best volume to send the duplicate copy. The user is responsible for both writing the duplicates and recovering synchronization after failure.

Block direct access datasets are the simplest dataset type, with contiguous records accessed by an ordinal record number. No control information is maintained concerning the usage of records within the dataset—this is the user's responsibility. The record size is fixed and equal to the record element size of the dataset extent which is specified when the file is created. As is common with all datasets, a record can be "locked" to permit a read-modify-write cycle to be performed safely in environments where a single dataset is opened by multiple users.

Mapped direct access datasets are similar to block direct access datasets except that these datasets use variable length records which may change in size during the course of an access. Each record created in a mapped direct access is identified back to the user on an individual record basis, the ID being used to reference the record on subsequent accesses. This technique allows multi-volume indexed datasets to be built, with an index-only dataset (see next section) on one volume, with its indexed memo entry pointing to the individual records of a mapped direct access dataset on another volume. These basic tools can be used to advantage in implementing more general structures such as multi-volume databases.

For all indexed datasets, records are referenced by an associated key held in a directory control area within the dataset. The key length is an attribute of the dataset and can be up to 255 bytes in length although in practice the smallest size possible is recommended. Either an exact or an approximate match can be specified in the access request. In addition to the basic read/write access of a record, it is possible to access the next record, to rename a key, to delete the record only or delete both the record and its key. To minimize data movement, a small amount of data can be stored with the key in the key index. This is termed a memo, and it can be read to or written from without accessing the whole of the associated record.

An index only dataset can be created that has no data area but contains only a key index. This allows the key index maintenance and access routines used with normal indexed type datasets to be used with nonsupported data structures created by the user. Verification and recovery of such structures is the creator's responsibility. The only meaningful data that can be transferred during the index-only dataset access is the content of the memo field. Accessing an index-only dataset returns a record ID of the index identical to that returned from accessing a mapped direct access datasets record. This allows hierarchies of linked indexes to be constructed.

An indexed integral dataset is the most general purpose dataset organization. Each individual record within the dataset can be of variable length (up to a maximum defined when the dataset was created) with a complete record transferred as a single operation.

In an indexed subfile dataset, each indexed record is equivalent to a direct access dataset. A basic record element is referenced via a key into the index of the dataset (this points to the beginning of the contiguous record elements making up the direct access subfile) plus an ordinal address within the subfile (this offsets the access to a point within the record elements making up the subfile).

To add greater flexibility to the indexed subfile, access to individual records in the subfile is directed through a map called the logical sequence array (LSA). A physical offset into the subfiles' record elements is made using the ordinal number provided in the access request after it has been mapped through the logical sequence array. By allowing the user to reorder and replace the values in the logical sequence array, the accession order of the physical records can easily be changed. Thus data can be manipulated by manipulating the logical sequence array rather than the real data records allowing the user to impose his own structure within the records of an indexed subfile type with minimum data movement. Each indexed subfile records has its own logical sequence array.

An indexed chain dataset is similar to the indexed subfile, except that there is no logical sequence array and the order of records in the subfile is not easily changed. The indexed structure is a set of non-contiguous physical records successively linked by pointers contained within each record element. Initial access to a record element is via the key index augmented with either the ordinal address of the physical record or with

the identity of a marker associated with a user-designated point in the set of physical records. Within an indexed chain dataset, a user can "mark" any individual record element using a value that is stored in a mark table associated with the record. Subsequent access to another record element in the same chain can then be requested by any of four references:

(1) Absolute ordinal location relative to the beginning of the subfile.

(2) A differential location relative to the last access.

(3) A reference by specific mark value.

(4) A specific number of marked locations, counted from the last access.

The indexed chain dataset is useful whenever access patterns are basically status. For example, a complete program library could be maintained as a single indexed chain dataset with each indexed entry a specific program code file, which is accessed serially.

THE APPLICATION ENVIRONMENT

A. Introduction

The system **100** is a general purpose multi-media computer system. The system is a tightly-coupled distributed network of multiple high speed processors, interconnected by a high-speed packet switching network and a fully distributed fault tolerant operating system that together provide a uniquely flexible processing environment. The system **100** is functionally organized into five subsystems each consisting of multiple processors performing specific functions (i.e., file handling, interactive processing, real-time, executive or general processing).

Application programs are supported by the information processing subsystem **906**. The information processing subsystem **906** may contain from 1 to 26 general purpose processors **942, 944** with each general purpose processor having up to 8 M-bytes of user memory. Application processing or development is supported by the general purpose processors, which can directly execute the pseudo-machine codes, or P-codes, generated by a high-level systems programming language such as Pascal. The P-code set is a high-level machine instruction set which is efficiently produced from block structured procedural languages such as Pascal, "C", ADA and FORTRAN.

The general purpose processors **942, 944** are supported by the other four subsystems for Input/Output, real-time, file storage and executive services. These systems utilize a system **100** assembler to generate SPM machine codes basic to all system **100** processors. To produce both high-level Pascal programs, which operate in a general purpose processor, and low-level SPM assembler programs to run in other system **100** processors, an interactive program development system (PDS) provides the system designer with the following facilities: (1) Pascal complier, (2) Pascal linker, (3) SPM assembler, (4) SPM linker, (5) Screen editor, and (6) programmer's utilities.

The Pascal language used with system **100** is the ISO Pascal with extensions. The extensions fall into two general categories:

(1) Those that enhance the basic capabilities of the language, for example, extra character manipulator routines not provided in the basic International Standards Organization (ISO) version, and

(2) Extensions that interface fo the Kernel system and lower level resources, allowing a Pascal program to call on the services of distributed hardware and software functions of the system **100**.

Within this environment, a complete multifunction application system utilizing all hardware/software resources of the system **100** can be realized as a set of programs coded in Pascal. Each Pascal program, when scheduled as an active P-code process in the system, runs in its own protected allocation of system resources called an ideal machine (IM). This allows low-integrity program development to occur along with production processing of highly critical applications, without compromising overall system integrity. These activities may occur concurrently in a single general purpose processor, simultaneously in multiple general purpose processors or a combination of both.

Each general purpose processor in system **100** has a resident executive system (REX) and an ideal machine monitor (IMM). REX contains programs to perform physical initialization, interrupting handling, event and process handling, memory management, I/O, list processing and various computational and utility routines. The ideal machine monitor provides programs to create ideal machines, program loading, initiation, scheduling and termination, extended memory management, ideal machine I/O interface, ideal machine program management and the interfaces to permit use of REX system services.

A general applications processor may have from 500K-bytes up to 8 M-bytes of user memory. Within this memory each process, operating as P-code, has private hardware-mapped virtual address space relative to other co-resident processes. Communication to other active processes in the system is through the ideal machine monitor. Within this environment, a large number of processes may be active concurrently. These could be multiple processes operating in a production mode, processes in development or checkout under the program development system or any combination of the two. Each program development system that is active within the system **100** requires an ideal machine resource from a general purpose processor and an interactive terminal. A systems designer can use the program development system to create source text files, compile or execute Pascal programs or assemble SPM programs. Pascal programs can be tested interactively from a program development system terminal whereas SPM programs require a systems console for checkout, and can be loaded for execution only by processes outside of the program development system.

Typically, a single P-code process would be sufficient to run an application using calls to functions provided by the Kernel system to satisfy its input, output and storage demands. If a multi-invocation, single function application is required (as will often occur, for example, in telephone answering support services), then the designer need only be concerned in programming the service for one terminal. This is because the ideal machine architecture used in the general purpose processor allows multiple invocations of the same function to be run concurrently with each innovation of the program supporting just one terminal. To avoid inefficient duplication of code, a single copy of an application's P-code is shared within a general purpose processor if more than one ideal machine is running the same function.

An applications designer will very rarely need to program at the SPM level, as the Kernel system provides access to all of the processing resources and sup-

ported peripheral devices. However, if an application calls for a new device type to be added, an SPM level-1 device handler must also be written. Programs at the SPM level execute as privileged tasks, thus a complete knowledge of system hardware and software architecture is needed to produce high-integrity programs that do not interfere with existing system processes.

B. Program Spaces

The system **100** provides two distinct programming environments—the host programming space and the image programming space. The host space represents the low-level programmability of the system **100** with programs resident in the program memory of a processor and encoded in SPM machine code. The executive services processors **916, 918**, the disk data processors **934, 936**, the interactive services executives **702, 706**, the real-time executives **406, 408**, and the real-time processors **410, 412** are all entirely programmed in SPM machine code, as are the programs which mechanize the ideal machines in the general purpose processors. Their program memories collectively form the host programming space of the system **100**. The resident executive (REX) of each processor is contained in this space as are hardware-coupled or speed-critical system components. For example, device and dataset handlers, real-time signal processing routines, the job scheduling, allocation and monitoring function, etc., are considered part of the host space. The SPM machine code set is speed-optimized with a basic instruction cycle time of 133 ns. Spare program memory capacity exists in many processors which can be used for any application processes needing to be written in SPM code. In particular, the real-time processors **410, 412**, when not being used to host dedicated functions such as Record/Playback, are totally available for user-specified real-time signal processing or other activities. If an application requires extra host space beyond that configured in a processor of a specific type, then additional processors of that type can be configured into the system **100**, up to the global maximum of 32 processors in a single system.

The host-space is not the normal applications programming environment. An intimate knowledge of the basic machine architecture of the SPM is required to utilize efficiently the large machine code set, and an application has to interface directly with REX for support services. Further, if the spare host space capacity of a minimum configuration system is used, then application processes would have to be co-resident with critical system processes, which would tend to reduce the overall reliability of the system. For although REX manages the program and data memories of a processor on an allocation basis, it does not provide any intrinsic memory protection as any such schemes would reduce the overall performance of the processor.

The image space provided in the general purpose processors provides an excellent applications programming environment. In the image space, each process resides in its own isolated, protected memory space and is implemented using a high-level instruction set, the code of which can be efficiently generated from source programs written in a systems programming language. The image space instruction set in a general purpose processor is implemented by using the program memory of the general purpose processor to host Pascal P-code instruction emulation routines, the actual high-level P-codes being fetched from the general purpose processors data memory. Thus, in a general purpose processor, the high-speed program memory is used in a fashion similar to a writeable microcode control memory and what is normally considered as data (user) memory in other processors is used to hold both P-code programs and process-related data.

General purpose processors are created by adding a general purpose extension unit to a standard processor module 500. P-code instruction emulation cycles are optimized by an extension SPM instruction, jump virtual, which vectors the SPM to a specific emulation routine via any one of four 256-way hardware instruction decode tables. The jump is based on the contents of a data memory location which contains the P-code to be executed. The data memory capacity of the general purpose processor may be extended to 8 M-bytes from the normal limit of 128K-bytes to allow a large number of co-resident P-code processes to exist and operate concurrently. Each P-code process maps its 64K-byte code space and separate 64K-byte data space via a set of 32 mapping registers to address physical memory locations in the 8 M-byte store range. To reduce process context switching time to a minimum, each process is assigned its own unique set of mapping registers from a pool of 128 sets of 32 that are available in each general purpose processor. These hardware features of the general application processor allow a large number of fast P-code processes to be supported with an effective P-code instruction cycle time of between 4 and 5 microseconds.

The ideal machine is a collective term used to describe the environment provided for a P-code process in a general purpose processor. It emphasizes the isolated, single-user resource that the general purpose processor provides for each resident P-code process, with an image machine architecture totally different from the underlying host general purpose processor. Each P-code process resides in its own ideal machine with 64K-bytes of P-code program space of unique virtual memory, which can map to shared physical memory, together with an unshared 64K-bytes of data space available for the stack and heap of the process while it is running. Both the code and data segments appear to be internally contiguous, and do not overlap or interfere with the code and data segments of other processes.

The P-codes in the code space are sequentially executed by a P-machine implemented as an SPM software emulator in a general purpose processor. The P-machine not only provides basic P-code interpretation and execution, but also provides access, via intrinsic calls, to the rest of the system **100**. A P-code process executes "sequentially" until suspended during an I/O or an intrinsic call, although in practice a general purpose processors CPU time is sliced on a "round-robin" schedule between all of the nonsuspended P-code processes resident in a general purpose processor.

An ideal machine environment of a P-code process includes P-code module overlap from disk, up to 64K-bytes of storage space, a stack and a heap (in inverted storage space opposite a stack). When an application process is being designed, no regard need be paid to the final physical organization of the process. The only restrictions being to keep the code generated to within 64K-bytes, and to ensure that stacks and heaps do not overlap when the program runs (i.e., do not exceed a total of 64K-bytes). Applications requiring code or data segments larger than 64K-bytes have recourse to a number of alternatives:

(1) Sequential processes can use the segment overlay facility provided by the ideal machine to swap in code segments from disk.

(2) Concurrent processes can create a network of P-code processes (or a mixture of P-code and SPM-code processes) that communicate via signals, packets, or shared memory files (if resident in the same general purpose processor).

(3) Processes requiring large data spaces can acquire unlimited amounts of free data memory by the memory file mechanism. This allows a data structure to be built which resembles a record-structured random access dataset which is memory rather than disk resident. If two or more P-code processes declare the same named memory file, they can use its shared records if both are resident in the same general purpose processor.

A P-code process can be used as a node in a jobs' process network in the same way as an SPM-code process. Once established, it can create further subordinate processes it creates.

In most applications, the primary node of a job will be a P-code process. Even if a job is real-time critical and requires most of the application to be SPM-code nodes, a high-level control program to initiate and orchestrate these processes will normally be written in Pascal to provide a structured and maintainable program. Applications which require a higher degree of information processing and altorithmic logic reside mostly in Pascal, using the access methods provided by the kernel system to communicate with and control devices and datasets. The majority of applications fall into this class due to the accessability of system **100** resources provided by the ideal machine monitor operating system. Most applications will not require additional SPM processes to be programmed. Such programming is needed only when a new device type is being interfaced to the system to provide the necessary software device manager.

Many of the level **2** kernel system functions are programmed in Pascal and reside in ideal machines in a general purpose processor. The system device manager (SYSDEV) which is responsible for assigning ownership and access rights to all devices attached to system **100** is an example of this. SYSDEV uses large, memory-resident tables to track the characteristics and changing usage of devices. These tables are examples of memory files. The large user 8 M-byte memory of the general purpose processor allows virtual machines to use such large data structures reducing the access frequency or disk storage and so increasing overall system performance.

A P-code process is initially created using the program development sys.tem to write and compile a Pascal program. The generated P-code module is linked with any required service routines, procedures or library modules to finally produce a load module. This load module is then entered into a program library from a system operator's terminal and linked to an entry in the system's function library. This gives the P-code program a standard program ID which can be used in a create job or create node request to the job scheduling, allocation and monitoring subsystem (JSAM). When JSAM retrieves the program descriptor from the library, it notes that this is a P-code process and so needs a general purpose processor with sufficient resources to build an ideal machine to host the process. Having selected a suitable general purpose processor, a create Pascal process request is passed to the ideal machine software in the general puppose processor. The ideal machine monitor software creates a new ideal machine for the P-code process and if the program is not already loaded, loads the code into general purpose processor data memory. The process is then dispatchable along with any other processes in their own ideal machine monitors.

JSAM can attempt to select a host general purpose processor **942, 944** which already has the program P-code loaded into data memory. The new process's ideal machine address space is then simply mapped out to the memory locations containing the code, sharing the same copy with any other processes running the same program. This reduces start-up time by avoiding a program load (although a load might still be required for process-related data modules), and reduces the amount of data memory needed for program space in identical, multi-process applications. An alternative process allocation strategy is to load-share across general purpose processors to improve the response time of the system. However, once each processor is running a copy of the same program code, code-sharing as described above would result from any further requests for the same program function.

C. Ideal Machine Monitor Service Functions

Programs resident in ideal machines interface to the rest of the system **100** using a set of ideal machine monitor routines. Each routine is called as a procedure from the Pascal program, and associated with each call is a parameter list of variables supplied by the user and resident in the user's own data space. Ideal machine monitor service routines are named as external in the users program, and linkage to them from the call in the user's program is made at link time.

The service functions provided by ideal machine monitor can be classified into the following areas:

(1) Input/Output to devices and datasets.

(2) Inter-process communications and process initiation.

(3) Event Management.

(4) Time Management.

(5) Memory Management.

The ideal machine monitor service routines either map into specific requests to the REX services available in the general purpose processor **942, 944**, or are performed directly by the ideal machine monitor subsystem itself (for example, general purpose processor memory management functions). Ideal machine monitor service routines that call equivalent REX functions generate no P-code in the user's code space other than a call to an ideal machine monitor intrinsic procedure. Other ideal machine monitor service routines will include some code in the user's code space that generates call sequences to basic REX functions.

I/O calls form the bulk of the services, and the standard interchange protocol established by REX is reflected in these procedures with the Acquire—Open—Access—Close—Release sequence required. However, specific routines are included to simplify the interface to the various dataset types that are supported on the system **100**. Thus, while many of the I/O calls can be equally applied to manage the data channels that exist between devices or datasets, a subset of these calls relate only to datasets.

The inter-process communication (IPC) routines allow IM processes to use the basic packet mechanism of the IPC network. A user process has access to primitive functions of the system **100** through these proce-

dures, in that any legal packet type can be constructed and dispatched by the IPC. Also included in these calls are routines to create, identify and destroy subprocesses. This allows the user to arbitrarily build complex process networks in the system **100**, consisting of both P-code and SPM-code processes using any available and allocatable system resource.

The event management routines enable a process to check for, and possibly wait on, a specified event. Events can be generated by both hardware and software and include interrupts, packet receipts, time-outs, I/O completion, process termination and signals from other processes.

Time management routines allow initiation of timers that can be used as the source of an event at some future time. This basic mechanism can be used to construct complex process scheduling activities.

Memory management services allow a process to obtain extra memory allocation from the large memory space of the general purpose processor. A unique Pascal variable type is accessed via the memory management routines—the memory file. To the user, this has the characteristics of a random-access file-type, but uses memory resident records rather than records stored on mass storage devices. The owning process has access to one record of the memory file at any one time, and the record window is accessed via the data space of the owning process. The memory file extends the data space available to a process without increasing its basic 64K-bytes of virtual address space. Furthermore, multiple processes can share the same named memory file providing an effective interprocess communications technique for ideal machines is resident in the same general purpose processor **942**, **944**.

The general nature of services provided to the user by the ideal machine monitor is closely related to the attributes of the system **100** as seen by the underlying REX services. However, the structuring power of the Pascal language enables these basic functions to be built into more complex mechanisms as needed, but still allows total control of, and access to, all the system **100** resources.

The ideal machine monitor input/output services routines (IOSRs) interface with the underlying REX input/output services routines to perform a requested I/O operation. The input/output services routines establish control and data paths between the user and the devices and datasets attached to the system **100**.

In general, an I/O operation proceeds as follows. A user requests that an I/O function be performed by calling the appropriate input/output services routines and supplying a set of parameters that define the details of the request. The ideal machine monitor input/output services routines function formats the parameters into REX input/output services routines call arguments and invokes the appropriate REX input/output services routines, or sequence of REX input/output services routines. During this time, the user's process is suspended until the operation is complete. The ideal machine monitor input/output services routines returns either with a successful completion or an error with the appropriate status to indicate why the operation was unsuccessful.

The exact sequence in which ideal machine monitor input/output services routines procedures need to be executed largely depends on the specific device or dataset being accessed. The ideal machine monitor input/output services routines procedures have been listed previously.

All devices which are acquired and all datasets which are open must have a unique file information block (FIB) associated with them. The file information block resides in the user's data space and contains descriptive information about the device or dataset. This information is used by the ideal machine monitor input/output services routines and must exist from Acquire to Release for devices and from Open to Close for datasets.

The access control block (ACB) is used to maintain information relevant to each particular access of a device or dataset. As such, at any given time, a unique access control block must exist for each active access. An access is established by the execution of one of the access type verbs (e.g., VREAD, VWRITE . . .) and remains active until the access is completed (usually by a VENDIO operation).

When a device is acquired, an acquire response message is returned which contains the reference information needed to open and access the device. To accommodate this, the ideal machine monitor input/output services routines maintains the message packets in its own environment with an associated unique identifier. This identifier is then placed in the user's file information block (FIB) for future reference as required by the ideal machine monitor input/output services routines.

Interprocessor communication in the ideal machine is accomplished by way of 16-word packets with the format of the packets left to the caller. The areas covered by these procedures handle the sending of packets to other processes, and also include the initiation, identification and termination of processes.

The following procedures and functions are available related to interprocess communications and management:

VSEND: Send a packet to a process.

VSIGNAL: Send a packet to a process within the same processor.

VCREATE: Create a new subprocess.

VQUIT: Terminate the calling process.

VCREATORPID: Return the PID of the creator of the calling process.

VDECLAREFC: Declare the valid packet function codes for the calling process.

VSELFPID: Return the PID of the calling process.

The event management procedures deal with the detection of events and not the allocation or deallocation of event control bocks (ECBs). To the user of the ideal machine, event control blocks are an internal structure used totally by the system to maintain events for the user.

The procedures and functions relating to event management include:

VCHECK-EVENT: Check for the occurrence of an event and return true if the vent has occurred, otherwise, return false.

VWAIT: Make the calling process non-dispatchable until an associated event occurs.

The procedures and functions furnished by the ideal machine related to time management include:

VSTART-TIMER: Start a REX timer for the current running process.

VCANCEL-TIMER: Cancel a timer that was set for this process

VTIME: Return the current time to the caller.

The resource manager is an internal ideal machine monitor function that handles the allocation, dealloca-

tion, and bookkeeping required to map the logical address space of the ideal machines into the 8 megabytes of physical memory in a general purpose processor. It does this by using 256 segment maps, with each map having 16 hardware registers, and each register mapping out to a single 4096 byte block. When a process is initiated in an ideal machine, a data segment map of 16 physical memory address registers is allocated for the processes' data segment. Then, if the program required to run the process is not already loaded, a code segment map is allocated and the program is loaded into the mapped area. After the maps have been allocated, physical memory pages (of 4096 bytes each) are allocated as required to accommodate the data memory requirements for the process. The first page of the Pascal stack is reserved as a process communication area and is used by the ideal machine monitor to manage the process.

In addition to the 64K-bytes of data memory that are available for the Pascal stack and heap, a process may use large memory files to extend the memory of the Pascal heap.

Two memory files are allocated for a single process. These files are managed on a record basis in a manner similar to direct access files on disk. Any number of such files (within general purpose processor memory limits) may be opened and, at any given time, any record of an opened file can be mapped into a set of pages on the heap of the user's data segment. The internal management of the data in a record is left to the user. These memory files may be shared by multiple users. However, memory files do not have to be shared in the same order by all users (i.e., Process "n" could have opened the files in the order D, C, B, A). Also, a memory file record can be mapped into more than one data segment at the same time and a single memory file can be opened multiple times by the same user.

The procedures and functions supported by the ideal machine monitor relating to memory management include:

VOPEN-MEM-FILE: Open a memory file for use by the calling program. This is allocated as the next 4K byte page on the heap.

VSEEK-MEM-RCD: This function loads the requested record in the map of the caller at the appropriate place as established by VOPEN-MEM-FILE.

VEXTEND-MEM-FILE: Extend the space allocated for the associated memory file by an additional number of records.

A set of utility procedures and functions is provided by the ideal machine monitor.

VGET-CREATE-PKT: Return the create request packet to the caller.

VLOG-IT: Print the contents of a packet on the local auxiliary extension board console.

D. Program Development System Overview

The program development system (PDS) is an interactive system: that allows the programmer to edit, compile and link programs directly from a terminal. From any of a number of program development terminals, a programmer can invoke the creation of a program development system to be run in an ideal machine. Each program development system provides the programmer with an environment similar to that of a stand-alone minicomputer that is totally dedicated to the programmer's own tasks. The programmer is also freed from the need to submit source programs to background batch-streams for compilation and linking. In addition, programs compiled for operation in ideal machines may be executed directly from the terminal within the environment of the program development system.

The program development system operates in a 64K-byte ideal machine and provides the following major facilities:

(1) Interactive Screen Editor
(2) Pascal Compiler
(3) SPM Macro Assembler
(4) SPM Link Editor
(5) Pascal Linker

The program development system provides a convenient, simplified interface to I/O services that can be used to manage user files from the program development system terminal. Text, data and code files can be copied and deleted using a simple dialogue, with text and data files printed with a single command. A listing of a user's dataset directory can be requested, and code files can be displayed and patched from the program development system terminal.

The program development system can be used to check out Pascal program modules prior to integration. The structured nature of the Pascal language and the strong type-checking of variables performed by the compiler reduces the normal checkout burden to verifying correct logical operation. To aid in interactive checking, the standard Pascal intrinsics READLN and WRITELN are available to be inserted in the program, with I/O directed to either the program development system terminal or the printer. This allows user-specified program tracing. In addition, the full range of standard run-time error-messages defined for Pascal are displayed at the program development system terminal.

These check-out facilities are not available once the program module is integrated into an application system running in its own virtual machine, as the program development system operating system that provides these services is no longer present. READ and WRITE are replaced by the comprehensive I/O services provided by ideal machine monitor input/output services routines.

The integration of SPM code into the host space of system **100** is a privileged task calling for a special means of debugging and testing, as well as responsible action on the part of the user. Native SPM code cannot be run directly under the program development system—only Pseudo-codes (P-codes) produced by the Pascal Compiler can execute in an ideal machine. To check out SPM programs, particularly those destined for ROMs or those requiring testing in a stand-alone processor, an auxiliary extension board (AEB) can be used to debug code in the target processor in which it is to reside. An auxiliary extension board, when plugged into a processor, provides complete control of an SPM via a soft "front panel", consisting of an interactive terminal and a printer. To the programmer, an SPM processor (together with a normal extension board) plus an auxiliary extension board workstation provide a complete stand-alone test and integration facility. In addition, a processor equipped with an auxiliary extension board can co-exist and participate in the normal operation of the system **100**.

For most SPM programs, checkout does not require an auxiliary extension board equipped processor. Instead, a system programmer's terminal **270** operating under the system monitor (SYSMON), can provide the programmer with the ability to initiate, terminate, monitor and alter SPM programs via the REX monitor.

When a new program function has been successfully tested, it is installed in the system program library from where it can be invoked automatically in response to external stimulus by the job scheduling, allocation and monitoring (JSAM) function of the kernel system. Insertion of a new program into the library is carried out from the system programmers terminal **270** through the system monitor. SYSMON also provides a program development console which can be used to supervise and manage all on-line program development systems.

E. Screen Editor and Utilities

Text files for input to either the SPM assembler or the Pascal compiler are generated and modified from a program development system terminal using the interactive screen editor. After invoking the editor, the user is asked to define a text file that exists in his directory, or alternatively, the name of a non-existent file that will be created as a result of the edit session.

The user is presented with a "window" into the text file with one terminal screen-full of data representing the current window position. The initial position of the window is at the top of the text file (or blank if a new file). The top line of the display continuously displays the current edit mode together with a list of the possible options that are associated with that mode. In this way, the user is always made fully aware of what options are available at any given stage in the edit.

The main edit mechanism is the cursor. The cursor can be moved around the screen and text can be inserted, deleted or change from the cursor position. Note that the screen of the program development system terminal is not the location where the actual text manipulation is taking place—this occurs in the program development system virtual machine. However, the high-speed link between the program development system terminal and the system maintains an accurate and updated image at the terminal of the current state of the text file being edited. By selective screen erasure and remote cursor control, the program development system achieves this without successive complete screen re-writes.

The following commands are available to the screen editor user:

Adjust: Adjusts the indentation of the line that the cursor is on. Uses the arrow keys to move. Moving up (down) will adjust the line above (below) by the same amount of the adjustment on the current line. Repeat-factors are valid.

Copy: Copies what was last inserted/deleted/zapped into the file at the position of the cursor.

Delete: Treats the starting position of the cursor as the anchor. Use any moving commands to move the cursor. "Etx" deletes everything between the cursor and the anchor.

Find: Operates in Literal or Token mode. Finds the target string. Repeat-factors are valid, direction is applied. "S"=uses same string as before.

Insert: Inserts text. Can use backspace and delete to reject part of an insertion.

Jump: Jumps to the beginning, end, or previously set marker.

Margin: Adjusts anything between two blank lines to the margins that have been set. Command characters can be used to protect text from being margined. (Invalidates the copy buffer).

Page: Moves the cursor one page in the current direction. Repeat-factors are valid; direction is applied.

Quit: Leaves the editor. Exit modes are Save, Exit, or Return.

Replace: Operates in Literal or Token mode. Replaces the target string with the substitute string. Verify option asks to verify before it replaces. S-option uses the same string as before. Repeat-factors replace the target several times. Direction is valid.

Set: Sets Markers by assigning a string name to them. Sets Environment for Autoindent, Filling, Margins, Token, and Command characters.

Verify: Redisplays the screen with the cursor centered.

Exchange: Exchanges the current text for the text typed while in this mode. Each line must be done separately. Back-space causes the original character to reappear.

Zap: Treats the starting position of the last item found/replaced/inserted as an anchor and deletes everything between the anchor and the current cursor position.

down-arrow: moves repeat-factor ("Repeat Factor" is a number typed before a control command. Typing a "/" repeates indefinitely.) lines down

up-arrow: moves repeat-factor lines up

right-arrow: moves repeat-factor spaces right

left-arrow: moves repeat-factor spaces left

space: moves repeat-factor spaces in direction

back-space: moves repeat-factor spaces left

tab: moves repeat-factor tab positions in direction

return: moves to the beginning of line repeat-factor lines in direction

" " "," "−": change direction to backward

" " "," "+": change direction to forward

"=": moves to the beginning of what was just found-/replaced/inserted/exchanged

A set of utility functions are provided by the program development system which allow the programmer to manage the files and programs generated by the program development system. A brief description of these utilities is as follows:

Hex Dump Allows the user to display and update disk files at the Hex code level. Options are:

Read: Read and display blocks of data in hex format.

Change: Change a byte of the displayed dump.

Update: Replace the updated block in the disk file.

Quit: Return to the editor.

Copy: Copy one file to another, with amendments if the destination file exists, or create a new file if no destination file is specified.

Delete: Delete a file from the user's directory.

List Directory: Displays or prints the user's file directory.

Print: Print a text or data file, either immediately if the printer is free, or spooled for later printing.

Execute: Load and execute a P-code file in the space above the program development system code in the ideal machine that hosts the program development system. Run-time errors trap back to program development system with a standard error message. Ideal machine monitor Input/Output services routines may be linked into the program. READLN and WRITELN are available to communicate with the program development system and provide user-defined program flow tracing.

F. SPM Functions

The SPM macro assembler generates relocatable SPM code segments from text file input. The object

code produced by the assembler can be input to the SPM link editor along with other object modules to produce a final relocatable load module.

The major features of the macro assembler are:

(1) Segmentation—The SPM program can be divided into numbered sections with the CSECT (Control Section) directive. The assembler concatenates the various components of a CSECT into a contiguous code section in the object module. Up to sixteen CSECTS can be defined in a program module, with one section having the attribute of COMMON. Common sections from all object modules are overlayed by the linker.

(2) Macros—User defined code sequences can be defined as named macros. Assemble time parameters can be passed to the macro expansion, allowing SPM programs to be structured out of tailored, functional components. Establishing common libraries of well tried macros is an efficient method of improving programmer productivity and program maintainability.

(3) Assembler Directives—The sequence of assembler processing can be controlled by directives in the text. In particular, the following two directives can be used to skip forward through text as determined by assembler evaluated expressions.

GOTO, Expression Symbol, Symbol, Symbol . . .

This skips to the next occurrence of a symbol, selected orinally from a list of symbols as determined by the evaluation of the expression in the GOTO.

JUMPVAL Expression **1**, Relation, Expression **2**, Symbol

Skips to "Symbol" if the "Relation" between 'Expression **1**' and 'Expression **2**' is true. These directives allow parameter-driven variants of a program to be assembled. For example, the changing of a parameter TEST from 1 to 0 could include or exclude test sections of code from a program.

Include from Text Library—A secondary text source can be defined as input to the assembler. This contains symbolically labeled sections of text that can be inserted into the primary source text stream when called by a LIBRARY directive. The format of the directive is:

LIBRARY, HOL(filename) Symbol **1**, Symbol **2** . . .

where "filename" is the name of the library file and "Symbol" identifies a section to be included.

Other directives are available with the SPM assembler. The assembler is a two-pass program that generates as output an assembly listing, a symbol dictionary and an object code file. The additional directives are:

Input Control:
COLUMN: Define continuation of source text line passed 79 characters.
LIBRARY: Insert from library file.
LEND: End of library insert.
Output Control:
PRINT: Switch output listing on.
PRINTOFF: Switch output listing off
PAGE: Restore listing to top-of-page.
SPACE: Insert blank lines
TITLE: Title and sub-title definition.
TABSET: Indentation of macro expansion listing.
LINES: Lines per page control.
OUTPUT: Allow output text from a macro expansion.
Location Counter:
ABSOLUTE: Defines absolute code location.
RELOCATE: Defines code location as relocatable.
ORIGIN: Set location counter.
RESERVE: Reserve memory locations.
BOUND: Set location counter to Modulo-n boundary.
CSECT: Following code belongs to numbered control section.
COMMON: Following code belongs to common control section.
Symbol Definition:
LABEL: Set an assembler symbolic label.
EQU: Define an expression equal to an argument.
LOCAL: Define a local (limited range) label.
SET: Redefine a symbol to an argument value.
SPNAME: Scratchpad symbol (as opposed to Prog. Mem.)
SPEQU: Scratchpad symbol using scratchpad address mode.
SPDEFAULT: Define address mode for all following SPEQUs.
RENAME: Rename a symbol (Both values available).
ALIAS: Rename a symbol (Only new value available).
Data Generation:
DATA: Set data in memory (various formats).
BYTER: Form one 16-bit word from two 8-bit bytes.
Program Module Communications:
DEFINE: Declare a global symbol.
REFER: Reference an external global symbol.
Input Statement Processing:
END: End of text input.
GOTO: Skip to computed symbol.
JUMPVAL: Skip to symbol on condition.
VOID: Conditionally skip macro expansion.
Macro Definition:
META: Define a macro.
MEND: Terminate a macro definition.
AMEND: Alternative macro exit.
Loop Control:
LOOP: Begin a repeated text insertion.
LOOP TEST: Conditional termination of repeat text loop.
LOOP EXIT: Unconditional termination of repeat text loop.

Object modules produced by the assembler are input to the SPM link editor. The output of the linker is a further set of modules that can be submitted to the relocating loader that loads program functions on behalf of the system. The linker allows the various CSECTS and COMMONS of the input object modules to be further structured and rearranged such that a logical sequential program is produced. A list of object modules (up to 16) is submitted to the linker. The user can then specify a particular CSECT value in the range of 0 to 15. The input modules are then scanned and all CSECTS with the specified value are extracted. The CSECT can contain both relocatable and absolute (i.e., fixed load address) code sections. An output load module is specified, and the relocatable CSECT components are contiguously placed into it. The loading into the output module of the CSECT can start at an offset from the beginning of the output module. Absolute sections of code are appended to the relocatable linked sections. When all the CSECT components have been transferred, a further CSECT can be specified.

COMMON sections are treated in a similar way except that each section is overlayed into the load module, i.e., the space in the load module taken by a given set of named common sections is equal to the size of the largest common in the complete set of common sections.

Any of 16 load module segments can be specified as the output location of the link, and multiple CSECTS and COMMONS can be directed to the same load segment. Any CSECT not specifically mentioned in a transfer is default loaded to segment **0**, and the default link is to link every object module in the order submitted into a single load module. Resolution of global references between the load module segments, and the assignment of relocatable values, finally takes place at load time. As programs are eventually loaded by REX into whatever memory resource is allocated (that is, not into known memory locations) the final relocation of a program's load module takes place dynamically at load time. Note that absolute program segments are normally used only for system functions such as REX, EXREX and related programs.

During the linking process, the user is able to supply undefined global references and can also reassign values to global references.

As output, the linker produces:

(1) A load module with up to 16 segments.

(2) A list of global references sorted by value in alphabetical order.

(3) A segment load map.

(4) A cross-reference of globals versus segments.

The following link editor commands are available:

CLEANUP: Define program entry point for orderly termination by the program.

EXTBASE: Set a default load module segment base.

EXTDEF: Set an undefined global reference, or override current value.

EXTEQU: Replace all references to global symbol "a" with value of symbol "b".

LIST: List the output load module.

MODULE: Specify beginning of input list of object modules.

OBJEND: End of list of object modules.

REFLIB: Library of reference values to be searched for undefined global references.

SECTION: Assign a CSECT to a load module section.

COMMON: Assign a COMMON to a load module section.

START: Define a normal entry point.

G. Extended Pascal

The Pascal language supported by the program development system is ISO Standard Pascal with certain extensions. Pascal is the major tool with which application systems are constructed. The Pascal compiler supported by the program development system generates Pseudo-Code (P-code) that is directly executable by the general purpose processor **942, 944** of the system **100**.

Pascal source text files are created with the screen editor and submitted to the compiler. The source text may contain compiler control switches as defined in Jensen & Wirth; however, the actual switches available for use with extended Pascal are different from those defined in the User Report. A monitor of progress of the compilation is shown on the program development system terminal, which displays numeric information relating to each procedure as it is compiled (Pascal is a single-pass compiler). If an error occurs, the line containing the error is displayed with the offending error highlighted. The user has the option of continuing with the compilation to highlight other errors, or entering the screen editor at that point to correct the error.

An include file mechanism is provided that allows text to be included from a source library. The use of the include mechanism allows large programs to be subdivided into segments for easier editing. However, once segments have been successfully compiled and tested, if is more efficient to link P-code segments together, rather than include the original text in every compilation.

Program development system Pascal compiler switches include:

C: Place comment in code file (ex. for copyright)

F: Change byte-se x of output code

G: Control use of "GOTO" statement in program

I±: Perform I/O check after each I/O operation

I(f): Include file "f" into source text

L(f): Send compiler listing to file "f"

M(n): Limit error messages to "n"

P: Top-of-form

Q: Console compite trace enable flag

R: Allow run-time check of array subscripts and variable subranges

The extensions to standard Pascal provided by extended Pascal are, in most cases, realized as procedure and function calls. The extensions can be divided into two classes—first, those that are satisfied by routines intrinsic in the ideal machine interpreter, and second, those that are satisfied by procedures in the program development system itself. The implication of this classified is discussed in the next section.

H. Pascal Linker

The two major areas addressed by these extensions are the provision of string functions for character manipulation, and extra file control and access routines.

One specific extension to standard Pascal is the SEGMENT PROCEDURE declaration. This allows a program to contain disk-resident procedures that are overlayed into the ideal machine when referenced from the program. In this way, a program can be sectioned into components that do not need to be resident concurrently in the ideal machine, allowing a program to be actually much larger than the 64K-byte code space available in an ideal machine, which is a size limit dictated by the 16-bit address range of P-codes. The net code size requirement is therefore the sum of the common, or root, segment of the program plus the size of the largest overlayed segment.

The following PDS Pascal extension procedures and intrinsics are available:

BLOCKREAD: A function that reads a variable number of blocks from an untyped file.

BLOCKWRITE: A function that writes a variable number of blocks to an untyped file.

CLOSE: Procedure to close files.

CONCAT: STRING intrinsic used to concatenate strings.

DELETE: STRING intrinsic used to delete characters from STRING variables.

EXIT: Intrinsic used to cleanly exit from the middle of a procedure.

GOTOXY: Procedure used for v.d.u. screen cursor-addressing whose parameters are column and line numbers.

FILLCHAR: Fast procedure for initializaing PACKED ARRAY's OF CHAR.

HALT: Halts a user program and return to Program Development System command mode.

INSERT: STRING intrinsic used to insert characters into STRING variables.

IORESULT: Function returning the result of the previous I/O operation.