MODULA AND THE DESIGN OF A MESSAGE SWITCHING COMMUNICATIONS SYSTEM

Gregory R. Andrews

TR 78-329

Department of Computer Science
Cornell University
Ithaca, NY 14853
MODULA AND THE DESIGN OF A MESSAGE
SWITCHING COMMUNICATIONS SYSTEM

Gregory R. Andrews
Department of Computer Science
Cornell University

This work was supported by the Scientific Services Program of Battelle Columbus Laboratories, Durham, and was authorized by the U.S. Army Research Office under Contract DAAG29-76-D-0100.
ABSTRACT

This report describes the functions of a message switching communications system and presents an implementation in terms of the Modula programming language. In particular, the report: (1) describes a representative application of the proposed new Department of Defense high order language; (2) presents a design technique for software specification; (3) develops Modula programs for each of the message switching components; and (4) evaluates the utility of Modula as a language for the design of large parallel systems.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2.0 System Specifications</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Hardware</td>
<td>4</td>
</tr>
<tr>
<td>2.2 System Functions</td>
<td>6</td>
</tr>
<tr>
<td>2.3 IO Interfaces</td>
<td>8</td>
</tr>
<tr>
<td>3.0 System Structure</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Summary of Modula</td>
<td>13</td>
</tr>
<tr>
<td>3.2 System Organization</td>
<td>17</td>
</tr>
<tr>
<td>4.0 Process Actions and Control Paths</td>
<td>26</td>
</tr>
<tr>
<td>5.0 Program Listings</td>
<td>34</td>
</tr>
<tr>
<td>5.1 Global Data Types</td>
<td>36</td>
</tr>
<tr>
<td>5.2 System Clock Group</td>
<td>38</td>
</tr>
<tr>
<td>5.3 MEMORY Interface Module</td>
<td>38</td>
</tr>
<tr>
<td>5.4 ACTIVE-ARCHIVE Module</td>
<td>52</td>
</tr>
<tr>
<td>5.5 IO Control - SWITCH Interface Modules</td>
<td>56</td>
</tr>
<tr>
<td>5.6 Subscriber Groups</td>
<td>61</td>
</tr>
<tr>
<td>5.7 Trunk Groups</td>
<td>81</td>
</tr>
<tr>
<td>5.8 SWITCH Process</td>
<td>93</td>
</tr>
<tr>
<td>5.9 Operator Group</td>
<td>110</td>
</tr>
<tr>
<td>5.10 System Initialization</td>
<td>121</td>
</tr>
<tr>
<td>6.0 Summary and Evaluation</td>
<td>127</td>
</tr>
<tr>
<td>Bibliography</td>
<td>135</td>
</tr>
</tbody>
</table>
1.0 Introduction

The unmistakable trend in recent years has been toward the use of high level languages for systems programming. In an effort to improve upon available tools, three new languages have been designed: Concurrent Pascal [1] and Modula [8] were developed to aid in the design and implementation of multiprogramming systems while Euclid [4] is intended for the programming of verifiable, sequential systems. All three borrow heavily from the work of Wirth in the design of Pascal [7]. Although intended primarily for the development of small operating systems, both Concurrent Pascal and Modula are applicable to parallel systems in general. In this paper, the design of one specific example, a message switching communications system, is developed and programmed in Modula. Our purpose is to show both (1) that a communication system can and should be viewed as a special purpose operating system and (2) that a language such as Modula is an ideal tool for its design. Modula was chosen as the target language because it is well documented, provides facilities for accessing machine hardware, and appears to be efficiently implemented [10].

At present, the Department of Defense is involved in a concerted effort to develop a new language (or family of languages) for use in implementing their software systems [2]. A message switching communications system is one example of such an application [5]; and Modula is a language
which meets many of the DoD requirements, specifically in
the areas of parallel processing and device control. A
message switch consists of a number of switching nodes, each
having local subscribers, connected via trunk lines. Its
function is to route messages from one subscriber to one or
more other subscribers connected to either the same switching
node or to another, remote switching node. Each switching
node accepts input messages from subscribers or trunks,
stores the messages on temporary storage and then forwards
complete messages to output destinations (either local
subscribers or trunks). This type of communication system
is often called a store-and-forward message switching system.

This report presents a detailed design in Modula of
the software for a switching node. (Each switching node
in a communications network would execute the same soft-
ware). In particular, the report:

(1) Develops one completely specified, typical
application of the proposed Department of
Defense language;

(2) Illustrates the use of top down design and
presents a descriptive technique for the
specification of parallel software systems; and

(3) Evaluates the utility of Modula for the
design of large parallel systems such as
the one presented.

Overall Modula proved to be a superb tool, although it
did present a few problems (enumerated in Chapter 6). As testimony to its power, the entire design described here was developed in thirty days (including the writing of this report). Much progress has been made since the early days of exclusive assembly language coding – and even if assembly language must still be used (in hopefully few places), we hope that this report provides justification for the use of a high-level language such as Modula as a tool for initial specification and subsequent documentation.

The next chapter gives the specifications of the hardware, user interface, and functions of the communication system. Chapter 3 contains a brief summary of Modula, and block diagrams of the communication system components. Before delving into detailed programs of each component, the different message processing phases are described in Chapter 4. Chapter 5 contains program listings and detailed descriptions of each component. Finally, Chapter 6 summarizes the design and evaluates the utility of Modula.
2.0 System Specifications

In this section, the hardware configuration and processing requirements of a typical message switching communication system are specified. The hardware is discussed at this point in order to give a feel for the size and nature of the switching system under consideration. The most important part of the specification of this or any system, however, is defining the formats of input and output message and the functions the system must perform on the messages (see [5] for more detail).

2.1 Hardware

The switching system considered consists of a network of switching nodes each connected to one or more other switching nodes via trunk lines. Connected to each node are a number of local subscribers, one special subscriber called the operator, archive tapes, and auxiliary memory for messages. A representative configuration is shown in Figure 2.1. Typically, a switching node has up to 50 subscribers and from 1 to 3 trunk connections to other nodes. Our design is independent of the number of subscribers and/or trunks, however. We also assume that there are four tape controllers, two for recording input messages and actions on the messages and two for retrieval of data or actions from previously processed messages. For temporary storage of input messages, each node has some auxiliary memory such as a disk.

Subscribers, as well as the operator, use terminals to interface to their local node. Each terminal is assumed to
Figure 2.1
Communication Hardware

subscriber
operator

switching node
archives
auxiliary memory

subscriber
operator

switching node
subscriber
archive tapes
auxiliary memory

trunk
trunk

subscriber: terminal
operator: terminal
switching node: processor
archives: tapes
auxiliary memory: drum or disk
trunk: communication lines
be a character oriented device, namely it transmits and receives data one character at a time. All terminals are full duplex so that input and output can proceed simultaneously.

Trunks are more complicated. Each consists of a bundle of pairs of sending and receiving lines. Information is transmitted along trunks in 84 character blocks. (The format of trunk messages is described in Section 5 when the trunk program is described.) Control signals are also passed along trunks to synchronize information transmission. We will assume that trunks transmit information at a periodic rate, namely information can be sent or received at fixed intervals rather than asynchronously as with other IO devices. Trunk lines are also assumed to be full duplex.

2.2 System Functions

The basic function of each switching node is to accept input from subscribers and route it to output destinations named in the headers of input messages. There are three phases involved in processing each message: input, switch, and output. In addition, the operator can request special functions, and is notified when exceptions are detected by the switching node.

The input phase involves receiving input from a subscriber or trunk, storing the message on auxiliary memory, and recording the input in an archive tape. Each message contains a header, a body, and an end marker. Once an
entire message has been stored, the switch function is invoked.

In the switch phase, two major actions are taken. First, acknowledgement of receipt of the message is recorded in an action archive and is output on the local operator's terminal. Second, the message header is examined to determine the output destinations. For each destination, the switch module selects an appropriate output line (either a local subscriber or a trunk) and starts the output phase. If a message is sent to more than one destination, each receives a copy. All messages are eventually output to subscribers (unless cancelled).

In the output phase, a message is retrieved from the auxiliary memory and transmitted to each destination. For trunk destinations, the message is sent as a sequence of blocks. For local subscriber destinations, the message is output as a sequence of characters. Each message contains a precedence and classification. At all times, the highest precedence message for each destination is output. If pre-empted, a message is later retransmitted in its entirety. The classification of each message is checked when it is output to a subscriber; it must be no greater than the current classification of the subscriber terminal.

Each switching node has one operator terminal. The operator can send and receive messages like any subscriber. In addition, the operator monitors and controls the activity
of the local node. The operator can request that the switching node perform certain actions (e.g. cancel a message, or retrieve a previously sent message). The operator also is notified of any exceptions or special actions occurring within the switching node (e.g. occurrence of a pre-emption, or need to mount an archive tape).

2.3 IO Interfaces

The main function of the switching node is to process input and output messages. The specific formats of subscriber and trunk IO messages are shown in Figures 2.2 and 2.3, respectively. Input header fields have the following values:

- **originator code** - the number of the line originating the message
- **destinations** - the number and identity of each intended output destination where the identity is a (switching node #, line #) pair
- **identification** - the date and time of input; together with the originator code this uniquely identifies a message
- **precedence** - emergency, routine, or deferred; emergency pre-empts the other two on output
- **classification** - classified or not classified (could use more levels in general)
- **local sequence number** -
- **subscribers** - value N meaning Nth input message
Figure 2.2
Subscriber Interface

Subscriber Input

header:  SOH code
         originator code
         destinations
         identification (date - time)
         precedence
         classification
         local sequence number
         EOH

body:    sequence of characters

end:     EOM code or
         cancel sequence, EOM code

Subscriber Output

header:  originator of message
         identification (date - time)
         precedence
         classification
         local sequence number

body:    sequence of characters

end:     EOM
Trunk Interface

Trunk Input

header: SOH code, SEL code
        originator
destinations
identification (date - time)
precedence
classification
list of switching nodes who have processed
        message
        ETX code, block parity

body: STX code, DEL code
      80 characters
      ETB code, block parity

end: ETX code, block parity (at end of block) or
      cancel sequence (inside block)

Trunk Output

header: same as above

body: same as above

end: same as above
of the day

trunks - sequence of numbers naming the switching
nodes which have processed this message

When a message is output, it basically contains the same
information as for input. Two differences exist for subscribers,
though. First, the destination no longer needs to be speci-
fied. Second, the local sequence number is changed to a count
of the number of output messages sent from the switching node
to the subscriber. We leave unspecified the actual length
and encoding of each header component; the program for the
system refers to fields by name.

Operators, as subscribers, can send and receive messages.
They have the same format as subscriber messages. In addition,
the operator of each node can make certain requests and receives
exceptions and other notices. The types of requests and
notices are enumerated in Figure 2.4. We assume that requests
have a starting code which enables them to be distinguished
from normal input messages.
Figure 2.4
Operator Interface

Operator Input

messages: same format as for subscriber input

requests: start of request code
body of request
key
values (up to four)
end of request code

key
"status"
"cancel"
"wait"
"restart"
"alter directory"
"alter line status"
"retrieve"
"cancel retrieve"

value(s)
types of status to retrieve
message identification
line number
line number
line number, new primary and
alternate destinations
line number, new status
type of retrieval, message
identification

Operator Output

messages: same format as for subscriber output

exceptions and notices: pre-emption, orbit, tape
mounts, cancellations, actions
on messages
3.0 System Structure

The message switch system has been programmed in Modula. This section briefly summarizes Modula and gives block diagrams of the system organization in terms of Modula constructs. Succeeding sections refine the structure into greater levels of detail.

3.1 Summary of Modula

Modula is a new programming language which is intended primarily for programming dedicated software systems. It is based on Pascal [7]. To the sequential language constructs of Pascal it adds two constructs for multiprogramming: processes and modules. It also allows the specification of so called device modules to control a computer's particular peripheral devices.

As an aid to the reader, a short summary of "sequential" Modula data types and control statements appears in Figure 3.1. Figure 3.2 summarizes the process and module constructs which will now be briefly discussed. For detailed information the reader is referred to Modula's defining document [8] and also to the excellent papers describing its use, design and implementation [9,10]. Our purpose here is merely to give the flavor of the language. In order to really understand the programs in Chapter 5, [8] should be consulted.

Processes have the same structure as procedures. Namely, they have parameters, local variables, and a set of statements. They are also activated in the same manner as procedures. The
Figure 3.1
"Sequential" Modula

DATA TYPES

Basic types: Boolean, char, integer, bits
Constants: `const` name = value
Types: `type` name = identifier | enumeration | array | record
Enumeration: (identifier list)
Array: array range of type
Record: record fields end
Variables: var names: type;...

PROGRAM STATEMENTS

Assignment: variable := expression
Procedure Call: procedure name (parameters)
If: if Boolean expression then statement list [elseif Boolean expr then statement list] [else statement list] end
Case: case expression of label_1: begin statement list_1 end ;
      ;
      label_n: begin statement list_n end end
While: while Boolean expression do statements end
Repeat: repeat statements until Boolean expr;
Loop: loop statements end
Loop exit: when Boolean expr do statements exit
difference is that when a process is "called," both the
process and the caller execute concurrently. In addition
to local variables, processes can access global variables and
call module operations.

Modules are like blocks in the Algol sense (they contain
declarations and statements). The difference is that a mod-
ule is a fence between the objects it declares and those
global to it. The purpose of a module is to make available
selectively those objects that represent an intended abstrac-
tion while hiding those objects that are considered details
of its representation. To specify the fence, a module con-
tains a define list and, optionally, a use list. The define
list names those objects exported from the module, namely
those objects accessible outside. Procedures, types, con-
stants, signals and read-only variables can be exported.
The use list names those global objects imported into the
module. If the use list is omitted, all global objects
are accessible; if it is present however, only global ob-
jects named in the use list are accessible. Modules also
contain statements to initialize local variables.

Two special types of modules play a key role for multi-
programming: interface modules and device modules. Interface
modules, which correspond to monitors [3], are modules which
provide exclusive access to defined procedures. If one process
is executing an interface module procedure, no other process
can execute within the interface module. Since it is usually
Figure 3.2

MODULA: Processes and Modules

**PROCESSES**

```pascal
process SWITCH;
variables
begin
;
end SWITCH;
```

**MODULES**

```pascal
module m;
define names; (*exported types, vars, procedures*)
use names; (*imported types, modules, etc.*)
declarations - variables, types, procedures
begin
    initialization
end m;
```

**INTERFACE MODULES**

modules with exclusive access to defined procedures
and with signal variables operated on by
wait(sig) and send(sig)

**DEVICE MODULES**

interface modules with internal device processes
which can have doIO statements to delay
execution until IO is complete
necessary for cooperating processes to synchronize their actions, interface modules can contain signal variables. Signals are sent by send(signal) and received by wait(signal). When a process waits for a signal it temporarily leaves the module thus relinquishing its exclusive control. A send results in a process switch if another process is waiting for the signal; otherwise it has no effect.

Device modules are special kinds of interface modules. In addition to defined, mutually exclusive procedures, a device module contains device processes. A device process, or driver, interfaces to the IO hardware of a specific device. Therefore, there is one device process for each addressable IO device. To represent time delays due to IO processing, device processes can contain doIO statements. Since device processes are within device modules, when executing they have exclusive access to the module's variables. Device processes relinquish control by waiting or by executing doIO. They regain control when signalled or when IO completes.

A Modula program is a module. Within this outer module all (non-device) processes and other modules are declared and the processes are activated. Process declarations cannot be nested, with the exception of device processes which are declared within device modules. Module declarations can be nested, however.

3.2 System Organization

There are five basic components in the message switch
system. At the center is the SWITCH process which directs activity and keeps track of the status of each message. Message input and output is handled by the trunk and subscriber components. The auxiliary memory group is used for storage of messages. The active-archive is used as a log of the data in messages and the actions taken on messages. Finally, the operator group processes messages (like the subscribers) and handles operator requests and system exceptions. The components and their interconnection are shown in Figure 3.3. The labels on the arcs indicate the types of operations which are performed. The trunk/subscriber groups, SWITCH process, and operator group direct activity. The auxiliary memory and active-archive groups provide services to the other three.

The specific organizations of each component, in terms of Modula units, are shown in Figures 3.4-3.7.* Trunks and subscribers are both organized in the same way; as will be seen in their programs (Chapter 5) they differ only as a result of the hardware difference between terminals and trunk lines. Their organization and interface to the rest of the system make differences transparent, however. Each consists of two controlling processes, one for input and one for output, as well as a device module for performing IO. Controller processes send or receive headers and blocks of data to and

*Capital letters are used for the names of the processes and modules; circles denote processes and boxes denote modules.
Figure 3.3
System Components
from the other components. The input control process in both cases reads from the input device and the output control process writes to the device. Within the device module are two IO drivers. Note that there are two control processes and one device module for each terminal or trunk line.

The trunk and subscriber input processes store a copy of each input block on the ACTIVE ARCHIVE. The ACTIVE ARCHIVE is implemented as a device module containing two driver processes, one for data and one for actions. The SWITCH process sends action messages to the ACTIVE ARCHIVE which in turn stores them on an action tape.

The auxiliary memory group has an interesting organization. Each message input to a subscriber or trunk is stored, by blocks, on auxiliary storage in a file created when input started. During output, the message is read back from auxiliary storage and, when output is completed, the storage file is destroyed. The MEMORY interface module provides operations (defined procedures) for create, destroy, read and write. For read and write it uses an internal device module to access auxiliary storage. This module schedules the IO operation which is in turn performed by a driver process.

The operator group also has an interesting organization. The main component is the operator subscriber which is the same as a normal subscriber (i.e. input and output control
Figure 3.4
Trunks and Subscribers

Figure 3.5
Active Archive
Figure 3.6
Auxiliary Memory

Figure 3.7
Operator
processes and a device module). In addition to handling normal messages, the operator receives exception messages and can request retrieval of past messages. To do so, the SUPERVISOR interface module is used. Exception messages are put into the SUPERVISOR module by the SWITCH process and are handled by the operator's output control process (SOUTPUT). Retrievals are initiated by the operator's input control process (SINPUT) and handled by the RETRIEVE process. The RETRIEVE process reads from the OLD ARCHIVE device module which contains driver processes for reading action and data archive tapes.

Input and output control processes interact directly with the ACTIVE-ARCHIVE and MEMORY interface modules. All message blocks pass between the control processes and MEMORY. Message headers, which contain all of the control information for a message, pass between the IO control processes and the SWITCH process. In addition, IO control processes and SWITCH exchange control signals. Interfacing the control processes to the SWITCH requires a number of interface modules. A diagram of the interface is shown in Figure 3.8. Since SWITCH is a process, it can only wait for one thing at a time. Therefore it receives all requests (e.g. new message, end of output, exception) from a NOTICE interface module. For new messages, it gets the header from LINEINPUT and stores it in HEADERS. At the end of input, it enters output directives for each destination line in LINEOUTPUT. Once
the output is scheduled, LINEOUTPUT retrieves the header from HEADERS and gives it to the appropriate output control process (one of the SOUTPUT or TOUTPUT processes). When output is complete, SWITCH receives a NOTICE, deletes the header from HEADERS and destroys the MEMORY file containing the message. The other interface module in Figure 3.8, REPLY, is used when input or output controllers need to wait for the SWITCH to respond to a notice. The flow of headers and control information between the IO controllers and SWITCH is described in more detail in the next chapter.
Figure 3.8
Trunk/Subscriber - SWITCH Interface

blocks to memory and archive

read message

SINPUT & TINPUT

end of input
headers

LINEINPUT

H

REPLY
responses
headers

LINEOUTPUT

NOTICE

update

write message

SOUTPUT & TOUTPUT

headers
control

blocks from memory

processes

interface modules
4.0 Process Actions and Control Paths

Before presenting programs for each process and module, in this chapter we summarize the actions of the main processes and describe the paths of messages through the system. The main actions of a switching node are centered in three areas: the input control processes, SWITCH, and the output control processes. Figures 4.1 - 4.3 summarize the functions of these three components in Shaw's flowchart-like notation called path descriptions [6].

For each message, an input process parses the header, groups the body of the message into blocks and finds the end of message code. After a header has been found, a MEMORY file is created and the header is sent to SWITCH via LINEINPUT. Each subsequent block of the message is stored in the MEMORY file and a copy is also stored in the ACTIVE ARCHIVE. When the end of the message is found, either a cancel or end of input NOTICE is sent to SWITCH.

SWITCH receives notices of many kinds (Figure 4.2). For now, three are important: header, end of input, and end of output (done). The others come from the operator (discussed shortly) or indicate exceptions. On receipt of a header NOTICE, SWITCH receives the header itself from LINEINPUT and stores it in HEADERS. SWITCH then logs an action messages on the ACTIVE-ARCHIVE and sends a REPLY to the input process which sent the header. When SWITCH receives an end of input NOTICE, it again logs an action message and then inserts one
Figure 4.1
Input Process Functions

repeat

Subscriber and Trunk Input: \(\rightarrow\) Build Header \(\rightarrow\) Build Blocks \(\rightarrow\) Process end of Message

Build Header: \(\downarrow\)
- Find next header component
- send header to LINEINPUT
- wait for REPLY

Build Block of Message \(\rightarrow\) Get next end of block char \(\rightarrow\) send block to ACTIVE-ARCH and MEMORY

End of Message
- cancel \(\rightarrow\) post cancel NOTICE
- close MEMORY \(\rightarrow\) post end NOTICE file
- normal
Figure 4.2

SWITCH Functions

SWITCH: -> receive NOTICE -> end of input ->
          -> done with output ->
          -> other notices ->

header: -> receive LINEINPUT -> enter in HEADERS -> put status in ARCHIVE -> send REPLY ->

end of input: -> put status in ARCHIVE -> insert in LINEOUTPUT for destinations

done with output: -> put status in ARCHIVE -> delete HEADERS -> destroy MEMORY file

other notices: ignore for now
output directive in LINEOUTPUT for each destination. When output is complete, SWITCH receives a done NOTICE. It again logs an action message, then deletes the header from HEADERS and destroys the MEMORY file containing the message body.

Each output control process (Figure 4.3) gets headers from LINEOUTPUT. On receipt of a header, it outputs the header and then outputs the body of the message by reading blocks from MEMORY and writing them on the output device (via the device module). While one message is being output, another message of higher precedence may be ready for output to the same device. When this happens, LINEOUTPUT sets a pre-emption flag. This flag is periodically examined by the output process and, if set, writing stops and the process receives the new header from LINEOUTPUT.

Figure 4.4 puts these three processes together. It shows the order of the actions taken by each component in processing any message from input through to output. The arrow on each arc indicates the direction of flow of information and synchronization signals.

The other major functions in the system are those of the operator. The operator handles both normal message input and output (in the same way as for subscribers and trunks) and special input requests and output messages. For retrieve, cancel, wait, restart, and alter requests, the operator posts a NOTICE for SWITCH. In the case of alter, the operator passes the new table values to SWITCH via the
Figure 4.3
Output Process Functions

Subscriber and Trunk Output:
- receive header from LINEOUTPUT
- output header
- output body of Message
- end

Output Header:
- change to output format
- write on device

Output Body:
- get block from MEMORY
- write on device
- end
- pre-emption
- get next header from LINEOUTPUT
Figure 4.4
IO Control - SWITCH Interface Timing

INPUT PHASE
1 send new header
2 post new msg NOTICE
3 receive NOTICE
4 receive header
5 store header
6 send REPLY
7 receive REPLY
8 store message on MEMORY
9 send end of msg NOTICE
10 receive NOTICE

OUTPUT PHASE
11 insert output msg in LINEOUTPUT
12 retrieve header
13 receive header
14 read message from MEMORY
15 send done with output signal
16 post done NOTICE
17 receive done NOTICE
SUPERVISOR interface module. When SWITCH receives one of these requests (via NOTICE) it takes care of it and sends a response back to the operator. These responses, as well as exceptions and special conditions (e.g. mount a new archive tape), are sent to the operator output process via SUPERVISOR. The control signal telling the operator that a response or other message is waiting comes from LINEOUTPUT in the form of a special header. Output processes get all their work from LINEOUTPUT in the same way that SWITCH gets all its work from NOTICE.
Figure 4.5
Operator Functions

Operator Input:
- get next request
- cancel
- wait/restart
- alter tables
- retrieve
- new tape

retrieve, cancel, → post NOTICE for SWITCH
wait/restart: → for SWITCH

alter tables: → send request to SUPERVISOR → post NOTICE for SWITCH

retrieve:
- start
  → send doretrieve to SUPERVISOR
- cancel
  → send cancel retrieve to SUPERVISOR

newtape: → tell archive tape is mounted

Operator Output:
- get operator output
- receive output
- write on notice from LINEOUTPUT
- msg from terminal
- SUPERVISOR
5.0 Program Listings

This chapter contains listings for each of the program components comprising the message switching node. Figure 5.1 gives an outline of the program. Succeeding sections of this chapter discuss each group in detail.

In order to make the listings more readable, three conventions which are not in Modula have been used. First, numeric constants are often denoted by a character string in quotation marks; for example "pre-empt" or "inheader." These could of course be represented in Modula by constant declarations. Upper bounds of arrays are usually just specified as "max"; the appropriate value to substitute is dependent on the actual size of the system.

Second, module procedures are called by specifying both the module name and procedure name. For example, SUBSCRIBER.read calls the read operation of the SUBSCRIBER module. In Modula, only the procedure name is used which requires that each procedure name is unique.

Third, queues have been added as an extension to the usual Modula data types. They are declared as:

```
name: queue maximum size of type
```

and can be operated on by three operations:
```
name.delete(entry)
name.insert(entry)
name.size
```

This shortcut has been employed within interface modules in order to decrease the length of the programs and, hopefully, increase their readability.

Queues as used here can be readily implemented in Modula in a variety of ways. The most obvious was is to use an array
module MESSAGE-SWITCH;

global data types - header, block, actionmsg, operator request, operator output

system clock group - TIMER device module, CLOCK process

MEMORY and ACTIVE ARCHIVE interface modules.

IO - SWITCH interface modules -
LINEINPUT, NOTICE, REPLY, HEADERS, LINEOUTPUT,
SUPERVISOR

Subscriber groups - SINPUT & SOUTPUT processes SUBSCRIBER device module

Trunk groups - TINPUT & TOUTPUT processes TRUNK device module

SWITCH process

Operator group - OPINPUT & OPOUTPUT processes
OPERATOR device module, RETRIEVE process
OLD-ARCHIVE device module

begin activate all processes

end MESSAGE-SWITCH
and three control variables as follows:

```plaintext
name: queue n of T becomes

name: array l : n of T;
    size, front, rear : integer;
    size := 0; front := 1; rear := 1;
```

The operations insert and delete then respectively add an element of type T at the rear of the array and delete an element from the front. The control variables are adjusted appropriately. The size operation of course just yields the value of size above. The four variables (array plus controls) could also be grouped into a record.

Another implementation is to use a module which defines the type of queue entries and the three operations insert, delete and size. Inside the module, queues are then represented as above by an array and control variables. Although this approach is feasible, a different module must be defined for each type of queue. Modula has no facility for so-called generic or polymorphic types. Since each of the queues used in the message switch system has a unique type the module approach would require a distinct module for each queue.

5.1 Global Data Types

Five types of data are used throughout the system: header, block, actionmsg, operator request, operatoroutput. They are declared in Figure 5.2. Type header contains all of the header and control information for messages. Block is the unit passed to and from MEMORY and the archives. An actionmsg is the type of information stored on the archive's action tapes. Operator request and operatoroutput define the formats of information passed to and from the SUPERVISOR module (which
Figure 5.2
Global Data Types

type header = record
origin: integer; (* line # of sender *)
outputcount: integer; (* no. of destinations*)
dests: array l: "max" of integer;
prec, class: integer; (* precedence, classification*)
identity: integer; (* date-time-origin*)
seqcount: integer; (* sequence data*)
sequence: array l: "max" of integer;
size: integer; (* no. of blocks in msg*)
filename: integer (* MEMORY file*)
end;

block = array l: "blocklength" of char; (* msg blocks*)

actionmsg = record (* actions stored on archive*)
msgid, time, action: integer end;

operatorrequest = record key: integer; (* type of request*)
value: array l:4 of integer end;

operatoroutput = record size: integer; (* no. of chars*)
data: array l: "max" of char end;
interfaces SWITCH to the operator).

5.2 System Clock Group

In order to keep track of the time of day and synchronize trunk IO, a TIMER device module and a CLOCK process are employed. The TIMER contains a device process, clock driver, which periodically receives (via doIO) a hardware clock interrupt. It then increments the time of day and sends a tick signal. The CLOCK process receives the tick signal and, for each TRUNK process waiting for IO synchronization (see Section 5.6), sends a trunk tick. If the period of the trunk tick is a multiple of the period of the hardware clock, the CLOCK process would accumulate clock ticks until the period has passed and then send a trunk tick. The TIMER and CLOCK are shown in Figure 5.3. The trunk and hardware clock periods are assumed to be equal for now. For an interesting discussion of hardware clock and Modula, the reader is referred to page 81 of [10].

5.3 MEMORY Interface Module

The MEMORY module provides an interface between IO control processes (subscribers and trunks) and auxiliary memory. It is organized as shown in Figure 3.6. A program outline of the module is shown in Figure 5.4 and the program is given in Figure 5.5. The MEMORY module manages free space on the auxiliary storage device and defines operations
Figure 5.3
System Clock Group

```plaintext
var trunktick : signal; (*synchronization for trunk IO*)
device module TIMER;
  define time of day, tick;
  var time of day : integer; (*number of hardware clock
                            interrupts since system initialization
    tick : signal; (*signal for each hardware clock interrupt
  process clockdriver;
    begin loop doIO; (*wait for interrupt*)
      inc(time of day); send (tick)
    end
  end clockdriver;
  begin time of day := 0; clockdriver
  end TIMER;

process CLOCK;
  use tick;
  begin loop wait (tick);
    while awaited (trunktick) do send (trunktick) end
  end
  end CLOCK;
```
Figure 5.4
Outline of MEMORY

interface module MEMORY;
    variables - directory, free space, waiting processes
    utility procedures - manage free space
    defined operations - create, write, endwrite, read, destroy

device module AUXMEM;
    variables - sector buffer, scheduling
    utility procedures - IO scheduling
    defined operation - IO
    process driver
        code to perform IO
    end driver;
    begin initialize AUXMEM
    end AUXMEM

begin initialize MEMORY
end MEMORY;
for managing files and performing IO. Five operations are
defined: create, write, endwrite, read, and destroy.

Create is called by input control processes. Its function
is to allocate storage space for the file and assign an in-
ternal name. The estimated size of the file is specified as
a parameter. If not enough space is currently available,
the user or sending trunk is notified and the input controller
waits. Once adequate space has been released (via destroy
or endwrite) the creating process continues.

Once created, a file is filled by calling write, speci-
fying the filename and data block. Write is in general called
many times. The write operation treats the file as a sequen-
tial file and writes into the next allocated external block.
Since auxiliary memory sectors are assumed to be larger than
data blocks, on most writes the old sector must be read, up-
dated, and then rewritten. Sectors of external memory are
allocated on demand and are linked together. The links are
stored in the directory of the file.

A file is "closed" by calling endwrite. The purpose
of endwrite is to tell the file system (i.e. MEMORY) how
much space was actually used. On creation, an estimate of
the maximum required space is specified, and MEMORY commits
that much space. Endwrite enables MEMORY to take back any
unused space and make it available for other files.

File reading is performed by input control processes
in subscribers and trunks. A call to the read operation
identifies the file, block number, and buffer to use.
MEMORY maps the file name and block number into a sector address and calls AUXMEM to perform the IO. The block number must be specified on read because many processes may simultaneously be reading different blocks from the same file (messages may have multiple destinations).

Once all processing on a file is completed, SWITCH calls destroy. Destroy frees the space occupied by the file and, if necessary, tries to awaken processes waiting to execute create. Waiting processes are awakened in the order in which they blocked regardless of how much space they need.

AUXMEM is a device module which schedules and performs read and write operations on auxiliary memory (if necessary). IO is actually performed by the driver process. The read and write operations in MEMORY give IO requests to AUXMEM by calling its IO operation. IO requests a turn, synchronizes with the driver process and then releases its turn. Request turn and releaseturn are scheduling procedures.

As defined in Figure 5.5, request turn and release turn use a first-come, first-served strategy. Other scheduling strategies, such as the elevator algorithm in [ ], can be readily implemented merely by changing the bodies of the scheduling procedures.

The driver process synchronizes with the IO procedure via startio and iodone signals. Notice that many processes could be in IO at once, waiting to be scheduled (by request turn). Only one process at a time can be waiting for iodone
however.

The code for MEMORY, which contains AUXMEM as a sub-module, is shown in Figure 5.5. AUXMEM is contained within MEMORY because it is part of the representation of MEMORY and hence is not directly accessible to control processes. MEMORY provides the abstraction of a file system. The abstraction, namely the file operations, are all that MEMORY's users see.
Figure 5.5
Auxiliary Memory Interface

interface module MEMORY;

define create, destroy, read, write, endwrite;

use NOTICE;

const sectorsize = n1; (* no. of blocks in sector*)
memorysize = n2; (* no. of sectors in memory*)
max # files = n3; (* maximum number of files*)

type file = record (* format of file descriptor*)
nname, claim, used, curblock: integer;
sectors: array 1: maxfilesize of integer
end;

var directory: array 1: max # files of file;
free directories: queue max # files of integer;
(* empty directories*)
free space: queue memorysize of integer;
committed: integer; (* no. of sectors claimed or
actually used *)
waitingdata: queue n of integer; (* queue of
requested sizes for processes waiting
to create *)
spacenowavail : signal; (* for processes waiting
to do create *)
i: integer; (* loop counter in initialization *)

procedure .spaceavail (size: integer): Boolean; (* size is
estimated no. of sectors *)

begin
  if committed + nsecs <= memory size
    then spaceavail = true
    else spaceavail = false
end .spaceavail;
Figure 5.5 (Continued)
Auxiliary Memory Interface

procedure request (sector, filename: integer);
begin
  freespace.remove(sector)
  with directory (filename) do
    inc(used); sectors(used):=sector end;
end request;

procedure release (filename: integer);
var i: integer;
begin
  i:=1; with directory(filename) do
    repeat
      freespace.insert (sectors(i))
      inc(i)
    until i>used
  committed:=committed-used end;
end release;

procedure create (msgid, size : integer; var filename : integer);
var nsecs : integer; (*estimated no. of sectors*)
begin
  nsecs := size div sectorsize;
  if size mod sectorsize > 0 then inc(nsecs) end;
  if freedirectories.size = 0 then (*do something about the exception - e.g. send
    NOTICE to SWITCH or wait for file to be destroyed*)
    end
  freedirectories.remove(filename);
  with directory (filename) do
    extname := msgid; claim := nsecs; used := 0;
curblock := 0
end
```pascal
if not spaceavail(nsecs)
  then
    NOTICE.post("stop", line$k) (*tell user to stop input*)
    waitingdata.insert(nsecs);
    wait(spacenowavail);
    NOTICE.post("restart", line$k) (*line # can be computed from msgid which contains the origin of the message*)
  end
committed := committed + nsecs;
end create;

procedure write(filename : integer; var buffer : block);
  var S : integer;
begin
  with directory(filename) do
  begin
    if curblock=0 then (*allocate new sector*)
      freespace.remove(s);
      inc(used);
      sectors(used) := S;
      AUXMEM.IO("write", buffer, S,0);
    else
      AUXMEM.IO("read/write", buffer, sectors(used), curblock);
    end
    inc(curblock);
    if curblock > sectorsize then curblock := 0 end
  end
end write;
```
Figure 5.5 (Continued)
Auxiliary Memory Interface

procedure endwrite(filename : integer);
    var allocate : Boolean;
begin
    with directory(filename)do
        committed := committed-claim+used (*update actual amount of committed storage*)
    end (* see if waiting processes can now proceed*)
    allocate := true;
    while allocate do
        if waitingdata.front <= memorysize-committed
            then waiting data.remove; signal(spacenowavail)
        else allocate := false end
    end
end endwrite;

procedure read(filename, blno0 : integer; var buffer : block);
    var S, O : integer;
begin
    with directory(filename)do
        S := blno div sectorsize; (*sector number*)
        O := blno mod sectorsize; (*offset in sector*)
        AUXMEM.IO("read", buffer, sectors(S), O)
    end
end read;
Figure 5.5 (Continued)
Auxiliary Memory Interface

procedure destroy(filename : integer)
  var allocate : Boolean;
  begin
    (*release file space*)
    release(filename);
    (* delete filename from directory*)
    freedirectories.insert(filename)
    (*awaken processes waiting to create files*)
    allocate := true;
    while allocate do
      if waitingdata.front <= memorysize-committed
        then waitingdata.remove; send(spacenowavail)
        else allocate := false end
  end
end destroy;

device module AUXMEM;
  define IO;
  use sectorsize

  var (*communication with driver process*)
  op,sec : integer; (*operation, sector*)
  IO avail : Boolean; (*operation ready for driver*)
  startio, iodone : signal; (*driver synchronization*)

  (*IO buffer for driver*)
  sectorbuffer : array 1 : sectorsize of block;

  (*variables for IO scheduling*)
  (*just use FCFS for now- could use elevator algorithm of Hoare*)

  turn : signal;
  deviceallocated : Boolean;
procedure requestturn(sector : integer)
begin
 (*schedule IO operations in an order which controls latency and rotation delays*)
 (*for now, will just use FCFS*)
 if deviceallocated then wait(turn);
 deviceallocated := true;
end requestturn;

procedure releaseturn(sector : integer)
begin
 (*select next process to get its turn doing IO*)
 deviceallocated := false;
 send(turn)
end releaseturn;

procedure IO(operation : integer; var buffer : block; sector, offset : integer);
begin
 requestturn(sector); (*wait to be scheduled*)
 case operation of
 "read" : begin op := "read"; sec := sector;
 IOavail := true; send(startio);
 wait(iDone);
 buffer := sectorbuffer[offset]
 end;

end;
Figure 5.5 (Continued)
Auxiliary Memory Interface

"write" : begin  
op := "write";  
sec := sector;  
sectorbuffer[offset] := buffer;  
IOavail := true; send(startio)  
wait(iDone)  
end

"read/write" : begin  
op := "read";  
sec := sector; (*read sector*)  
IOavail := true; send(startio);  
wait(iDone);  
sectorbuffer[offset] := buffer; (*update sector*)  
op := "write";  
sec := sector; (*write it back*)  
IOavail := false; send(startio);  
wait(iDone)
end; (*of case*)  
releaseturn(sector) (*let next process be scheduled*)  
end doIO;

process driver;

begin
loop  
if not IOavail then wait(startio);
format operation on sector into or out of sector buffer;
doIO;

IOavail := false; send(iDone)
end
end driver;
begin (*initialize AUXMEM*)
    IOavail := false; deviceallocated := false;
    driver
end AUXMEM;

(*initialization of MEMORY*)

begin
    (*initialize all directories to free*)
    i := 1
    repeat freedirectories.insert(i)
        inc(i)
    until i > max#files;
    (*initialize free space*)
    i := 1
    repeat freespace.insert(i)
        inc(i)
    until i > memorysize
end MEMORY;
5.4 ACTIVE-ARCHIVE Module

The ACTIVE-ARCHIVE module is organized as shown in Figure 3.5. It provides an interface to the archive tapes. All data blocks are stored on a data tape; all actions taken on a message are stored on the action tape. To cause data and action messages to be written, the ACTIVE-ARCHIVE provides two operations, data and action (there are actually three - resume is discussed below). Both operations store their parameters (a block or action message respectively) in a buffer. When the buffer is full, it is output by either the data archive or tape archive device process. For efficiency, namely to reduce the space taken up by inter-record gaps, messages are blocked before transmission to the tape. Neither blocks or action messages are ordered by the sender; data or action messages from different input devices are in general interleaved. Each has an identifier field however, in case it ever needs to be retrieved (see Section 5.9).

The ACTIVE-ARCHIVE program is listed in Figure 5.6. Its logic is straightforward. The one exception occurs when a tape has been filled. In this case, SWITCH is notified (via NOTICE). SWITCH will subsequently tell the operator to mount a new tape. Once it is mounted, the operator's input process calls resume which allows writing to continue.
device module ACTIVE-ARCHIVE;

define action, data, resume;

use NOTICE, block, actionmsg;

constant actiontapesize = m_1; (*# of records on action tape*)
datatapesize = m_2; (*# of records on data tape*)
actionrecordsize = n_1; (*# of msqs in action record*)
datarecordsize = n_2; (*# blocks in data record*)

var arno, drn0, abn0, dbn0 : integer; (*current count of active data records and blocks*)

(*declare blocking buffers for tapes*)

actionbuffer : array 1 : actionrecordsize of actionmsg;
datapufer : array 1 : datarecordsize of record id : integer info : block end;

outputaction, actiondone, outputdata, dataone,

tapemounted : signal; (*driver synchronization*)

actionavail, dataavail : Boolean;

procedure action (act : actionmsg);

begin (*write actionmsg on action tape*)

inc(abn0); (*store action message*)

actionbuffer(abn0) := act;

if abn0 = actionrecordsize

then actionavail := true; (*output action buffer*)

signal(outputaction);

wait(actiondone);

abn0 := 0;

inc(arno)

end;
Figure 5.6 (Continued)
ACTIVE ARCHIVE

\begin{verbatim}
if arnO = actiontapesize (*end of tape*)
then (*tell operator to mount new tape*)
   NOTICE.post ("exception", "mountactiontape");
   wait (tapemounted);
   arnO := 0 end

procedure data (insgid : integer; bl : block); (*write msgid and block on data tape*)
begin
   inc(dbnO); (*store block*)
   databuffer(dbnO).id :=msgid;
   databuffer(dbnO).info := bl;

   if dbnO = datarecordsize
      then (*output data record buffer*)
         dataavail := true;
         signal (outputdata);
         wait (datadone);
         dbnO := 0;
         inc(drnO) end

   if drnO = datatapesize (*end of tape*)
      then (*tell operator to mount new tape*)
         NOTICE.post ("exception", "mountdatatape");
         wait (tapemounted);
         drnO := 0 end

end data;
\end{verbatim}
Figure 5.6 (Continued)
ACTIVE ARCHIVE

procedure resume;
(*called by OPERATOR INPUT process when operator says that
a new tape has been mounted*)

begin
  signal (tapemounted)
end resume;

process dataarchive;

begin loop
  if not dataavail then wait(outputdata) end;
  initiate output of contents of data buffer;
  doIO;
  dataavail := false;
  signal (datadone)
end loop
end dataarchive;

process action archive;

begin loop
  if not actionavail then wait (outputaction ) end;
  initiate output of contents of action buffer;
  doIO; actionavail := false;
  signal (actiondone)
end loop
end action archive;

(*initialize device module*)

begin
  arnO := 0; drnO := 0; abnO := 0; dbnO := 0;
  dataavail := false; actionavail := false
  dataarchive; actionarchive
end ACTIVE-ARCHIVE;
5.5 IO Control - SWITCH Interface Modules

As shown in Figure 3.8, the subscriber and trunk IO control processes interact with SWITCH via five interface modules: LINEINPUT, NOTICE, REPLY, HEADERS, and LINEOUTPUT. Modula programs for each of these modules are given in Figures 5.7 - 5.11.

LINEINPUT is shown in Figure 5.7. It defines two operations, sendhead and receivehead. Sendhead is called by input control processes; it stores a header in the headers queue and posts a NOTICE to tell SWITCH that a new header has arrived. SWITCH calls receivehead once it receives the NOTICE; it returns the first header. LINEINPUT acts like a simple message passing module except that because SWITCH only calls receivehead when it knows a header is available, receivehead never causes SWITCH to wait.

The NOTICE interface module implements a bounded buffer of notices for SWITCH. Notices are posted from a variety of places whenever SWITCH needs to be told something. They are only received by SWITCH, however. The program for NOTICE is given in Figure 5.8. Each notice has a kind field to tell what kind of data it contains. In Section 5.8 the different
interface module LINEINPUT;

define sendhead, receivehead;

use NOTICE, header;

var headers : queue "max\$" of header; (*sent headers*)
nonfull : signal; (*synchronization*)

procedure sendhead (hd : header);
begin
  if headers.size = "max\$" then wait(nonfull) end;
  headers.insert (hd);
  NOTICE.post ("head",0)
end sendhead;

procedure receivehead (var hd : header);
begin
  header.delete (hd);
  signal (nonfull)
end ;

begin (*headers is initially empty*)
end LINEINPUT;
interface module NOTICE;

define post, receive;

type note = record k, d : integer end;

var nonempty, nonfull : signal; (*synchronization*)
notices : queue n of note; (*pending notices*)

procedure post (kind, data : integer);

var n : note;

begin if notices.size = n then wait (nonfull) end;
  n.k. = kind; n.d. = data;
  notices.insert(n);
  send(nonempty)
end post;

procedure receive (var kind, data : integer);

var n : note;

begin if notices.size = 0 then wait(nonempty) end;
  notices.delete(n);
  kind := n.k; data := n.d;
  send (nonfull)
end receive;

begin (*notices is initially empty*)
end NOTICE;
values for kind and data are enumerated when SWITCH is discussed.

REPLY is similar to NOTICE and is shown in Figure 5.9. The main difference is that many processes can receive replies; in particular all input and output control processes wait at times for a REPLY. Consequently, REPLY uses an array of signals, one per line number. The receive operation returns an integer data value once it is available. Replies are sent by calling the give operation and specifying the line number and data.

The fourth interface module, HEADERS, provides storage for the headers of all active messages. Its program is shown in Figure 5.10. When SWITCH receives a new header from LINEINPUT, it calls HEADERS.enter. Enter selects a free header "slot" and stores the header in it. The index of the selected slot is returned to SWITCH and becomes the internal identifier of the message. As the message corresponding to the header is processed, the header is occasionally updated by calling retrieve, changing some values, and then calling update. Once the message has been output or cancelled, the header is destroyed by calling delete. Initially all header slots are put on the free queue.

The final interface module connecting IO control processes to SWITCH is LINEOUTPUT, shown in Figure 5.11. Its functions are to schedule and control output activity. For each output line, LINEOUTPUT has a linequeue record which is the header of a list of output messages for that line. For each
interface module REPLY;

define give, receive;

var replies : array 1 : "#lines" of integer;
available : array 1 : "#lines" of signal;
i : integer;

procedure give (line, data : integer);
  replies [line] := data;
  send (available [line])
end give;

procedure receive (line : integer; var data : integer);
  if replies[line] = 0 then wait (available[line]);
data := replies[line];
replies[line] := 0
end receive;

begin i := 1
  repeat replies[i] := 0; inc(i) until i > "#lines"
end REPLY;
interface module HEADERS;
define enter, retrieve, update, delete;
use header;

var hd : array 1 : "max $\#$" of header; (*full headers*)
   free : queue "max$\#$" of integer; (*empty header slots*)
i : integer;

procedure enter (h : header, var index : integer);
   var i : integer;
   begin if free.size = 0 then error end;
       free.delete(i); index := i;
       hd(i) := h end enter;

procedure retrieve (var h : header; index : integer);
   begin h := hd[index] end retrieve;

procedure update (h : header; index : integer);
   begin hd[index] := h end update;

procedure delete (index : integer);
   begin free.insert(index) end delete;

begin i := 1; (*initialize free list*)
   repeat free.insert(i); inc(i) until i > "max$\#$"
end HEADERS;
output message, the kind of message, its internal name
(HEADERS index) and precedence are stored. The msgs array
is the storage area for all output messages. Free message
slots are kept in the free queue. Each line queue stores
messages in decreasing order of precedence. That is, the
highest precedence output message is always kept at the head
of the list. Within any precedence level, output messages
are ordered by time of arrival. The other main variables
are a Boolean array of preemption flags set by insert
(discussed shortly) and an array of available signals used
to synchronize output control processes.

LINEOUTPUT provides four operations: insert, receive,
done, and cancel. Insert is called by SWITCH, once for each
destination of an input message; it adds an outputmsg to
the appropriate output queue. If the output queue is empty,
the message goes at the front and available is signalled.
If the new output is of higher precedence than the one at
the front of the queue, the new message is put at the
front and the pre-empt flag for the line is set. This will
cause the appropriate output control process to stop and
call LINEOUTPUT to receive the new, high precedence message.
If the new message is of equal or lower precedence than the
one at the front of the queue, it is inserted at the appro-
priate place.

The receive operation is called by output control
processes whenever they are ready to output another message.
If none is available, the process waits. When a message is,
available the output controller receives the header of the first output message on the linequeue. Some special messages, which are merely directions to output processes, are also sent via LINEOUTPUT. Since these do not have headers, only a kind indicator is returned. Note that received messages remain on the linequeue. In this way they can be received again if pre-empted by higher precedence output.

Once output is complete, the output control process calls done. Done merely deletes the first entry on the linequeue and for regular messages (those having headers) notifies SWITCH. For simplicity, done does not return the next available message if there is one. The output controller gets the next one via receive.

The final operation is cancel. It is called by SWITCH in order to cancel the output of a message (when directed to do so by the operator). Because a message may be sent to more than one destination and may be in different stages of output to those destinations, cancel merely marks the message by setting kind to "cancel". Eventually each output destination controller will receive the message, process the cancellation and call done.

5.6 Subscriber Groups

Each subscriber group provides an interface to a user terminal. The organization of each group is the same and was shown in Figure 3.4. The programs for an SINPUT and an SOUTPUT process as well as a SUBSCRIBER device module
Figure 5.11
Line Output Queues

```plaintext
interface module LINEOUTPUT;
    define pre-empt, insert, receive, done, cancel;
    use HEADERS, NOTICE, header;
    type outputmsg = (*output control information*)
        record kind, index, pr : integer;
            link : integer; end;

    linequeue = (*list header for line*)
        record front, rear, size : integer; end;

    var msgs : array l : "max" of outputmsg (*storage for output messages*)
    free : queue "max l" of integer; (*free message slots*)
    outqueue : array l : "#lines" of linequeue; (*queues of available messages*)
    pre-empt : array l : "#lines" of Boolean; (*pre-emption flags*)
    available : array l : "#lines" of signal; (*output synchronization*)
    i : integer;
    doinsert : array l : "#lines" of signal; (*pre-emption synchronization*)

procedure insert (line, msgkind, msgindex, precedence : integer);
    (*insert output message on outqueue (line) at appropriate precedence*)

    var slot : integer
    begin with outqueue(line) do
        free.delete(slot) (*get empty slot-fill in values*)
        with msgs[slot] do
            kind := msgkind; index := msgindex;
            pr := precedence, end
```
Figure 5.11 (Continued)
Line Output Queues

if pre-empt [line] then wait (doinsert[line] end; (*avoid pre-emption conflict*)
if size = 0 (*empty linequeue*)
    then front := slot; rear := slot;
    size := 1; msgs[slot].link := 0;
    send(available[line]) (*wake up output process*)
elseif precedence > msgs[front].pr (*pre-emption*)
    then (*put new message at front*)
        msgs[slot].link := front;
        front := slot; inc(size);
        pre-empt[line] := true;
        NOTICE.post ("pre-empt", msgindex) (*inform SWIT of pre-emptic:
else (*put new message at appropriate spot in linequeue*)
    i := front;
    while i ≠ 0 and precedence <= msgs[i].pr do
        j := i; i := msgs[i].link end;
        if i = 0
            then (*insert at end of linequeue*)
                msgs[slot].link := 0;
                msgs[rear].link := slot;
                rear := slot
            else (*insert in middle*)
                msgs[slot].link := i;
                msgs[j].link := slot
        end
    inc(size)
end (*of conditional*)
end (*of with*)
end insert;
Figure 5.11 (Continued)
Line Output Queues

procedure receive (line : integer; var knu ; integer; var hd : header);
("fetch first output message from out queue (line) as soon as one is available")

var i : integer;

begin with outqueue[line] do

if size = 0 then wait(available[line]) end;
("retrieve first message")

knd := msgs[front].kind

if knd = "newmsg" or "acknowledgeinput"
then HEADERS.retrieve (hd, msgs[front].index)

end

pre-empt[line] := false;
send (doinst[line]);(*let another pre-emption*)
("occur if one is pending")

end

end receive;

procedure done (line : integer);
("called when last received output message is finished")

var f; i : integer;

begin with outqueue[line] do

("delete first entry on queue - if pre-empt [line] then
 delete second entry")

f := front;

front := msgs[front].link;

if pre-empt[line] (*a pre-emption insert has occurred but has not been recognized*)
then f := front; front := msgs[front].link end

free.insert(f);
dec(size)

if rear = f then rear := 0 end
Figure 5.11 (Continued)

Line Output Queues

(*tell SWITCH output is complete for regular messages*)
\[ i := \text{msgs}[f].i\text{ndex}; \]
\[ \text{if } i > 0 \text{ then NOTICE.post("done",i) end} \]
end
end done;

procedure cancel (ind : integer);

(*find and mark any output messages identified by ind - the message may be in more than one linequeue - for each copy of the message set kind to "cancel; and, if it is at the front of the queue, set the pre-empt flag*)

var i : integer; hd : header; cnt, ptr : integer;
begin
HEA\-DERS.retrieve (ind,hd)
cnt := 1;
repeat (*for each output destination in header*)
\[ i := \text{hd.de\-ests}[cnt]; \] (*line of destination*)
with outqueue[i]do
ptr := front;
while ptr <> 0 do
\[ \text{if } \text{msgs}[ptr].i\text{ndex} = \text{ind} \]
then \[ \text{msgs}[ptr].\text{kind} := \text{"cancel"} \]
\[ \text{if } \text{ptr} = \text{front} \text{ then} \]
\[ \text{pre-empt[i]} := \text{true end} \]
end
\[ \text{inc(cnt)}; \]
until cnt > hd.outputcount
cancel;
begin (*initialize LINEOUTPUT variables*)
    i := 1;
    repeat pre-empt[i] := false
        until i > "#lines";

    i := 1
    repeat free.insert(msgs[i])
        until i > "max"

end LINEOUTPUT;

are shown in Figures 5.12 - 5.14. Because Modula does not provide any means to declare processes or modules as types, an actual system must contain one group of these three components for each subscriber terminal.

An SINPUT process takes the actions shown in Figure 4.1. Its role is to read characters from a terminal (via its SUBSCRIBER device module) and group the characters into headers and blocks. The program is shown in Figure 5.12. SINPUT is organized as a loop repeated for each character. First the character is read and then an action is taken depending on the current status of input. There are three states: find start, in head, and in body.

When no message is being processed, SINPUT is in "find start" status and looks for a start of message sequence of characters. Once the start has been found, status changes to "in head". Subsequent characters are parsed (detailed code is not shown since it depends on the exact message format) and a header is built. When the end of the header input is detected, a memory file is created, the header is archived, the header is sent to SWITCH (via LINEINPUT) and status is changed to "in body". The character sequence comprising the body of the message is then processed. For each block, MEMORY.write and ARCHIVE.data operations are called. Once the end of the message is found, the last block is taken care of and SWITCH is notified. If the message is cancelled, SWITCH is told to cancel input. After detecting the end of
process  INPUT;  (*one INPUT process for each line*)
     use  header, block, blocklength, SUBSCRIBER, LINEINPUT,
          REPLY, MEMORY, ACTIVE-ARCHIVE, NOTICE, HEADERS;

var  status : integer;  (*where INPUT is in the message*)
     bl : block;  (*buffer for building input message*)
     hd : header;  (*header built for new input*)
     name : integer;  (*internal msg identifier*)
     msgid : integer;  (*external msg identifier*)
     filename : integer;  (*name of MEMORY file*)
     current : integer;  (*character pointer into bl*)
     nblocks : integer;  (*number of input blocks in a message*)

begin  (*initialize variable*)
    status := "find start";

loop

SUBSCRIBER.read(ch);  (*get next input char*)

    case status of

"find start":  begin

       look for start of message sequence of characters;
       keep record of where you are in sequence;

       if entire SOM sequence has been received

          then current := 1;
          status := "in head"

       end

end;  (*of find start case*)
Figure 5.12 (Continued)
Subscriber Input Process

"in head": begin
(*store character received*)
bl.[current] := ch;
inc(current);

find next header component in bl;
store it in hd;
if error then status := "find start"
NOTICE.post ("exception", data) end;
if end of header found then (*tell SWITCH*)
LINEINPUT.sendhead(hd);
REPLY.receive (line# i, name); (*name is internal name of message assigned by SWITCH*)

(*create an auxiliary storage file*)
MEMORY.create (hd.identity, hd.size, filename);
(*filename is assigned by MEMORY*)
nblocks := 0;
ACTIVE-ARCHIVE.data( msgid, bl);
(*set current to start of block and start reading of body of message*)
current := 1; msgid := hd.identity;
status := "in body"
end (*of end of header condition*)
end; (*of in head case*)

"in body": begin
(*store character received*)
bl(current) := ch; inc(current);
(*look for end of msg(EOM) or cancel sequence*)
if end of message then fill rest of bl with blanks;
(*write block on ARCHIVE And MEMORY*)
ACTIVE-ARCHIVE.data (msgid, bl);
MEMORY.write (filename, bl); inc(nblocks);
MEMORY.endwrite (filename); (*close file*)
(*store actual size and file name in header*)
HEADERS.retrieve (hd, name);
hd.size := nblocks;
hd.filename := filename;
HEADERS.update(hd, name);
Figure 5.12 (Continued)

Subscriber Input Process

(*notify SWITCH of end of input*)
NOTICE.post ("end of input", name);
(*reinitialize*)
status := "find start"

elseif cancel input sequence found then
(*archive block and notify SWITCH*)
ACTIVE-ARCHIVE.data (msg,bl);
NOTICE.post ("input cancel", name)
status := "find start"

elseif current = blocklength then (*write out block*)
ACTIVE-ARCHIVE.data (msgid,bl);
MEMORY.write (filename, bl); inc(nbblocks);
current := 1
end

end (*of in body case*)
end (*of loop*)
end INPUT;
message or cancel, SINPUT sets status to "find start" and repeats the above actions.

The program for an SOUTPUT process is shown in Figure 5.13. SOUTPUT also executes as a loop receiving an output command from LINEOUTPUT, completely processing the command and then repeating the process by getting another LINEOUTPUT command. (See Figure 4.3) There are six kinds of output commands: new message, acknowledge input, cancel, stop input, restart input, and stop output.

New message is the most common type of command. It is sent whenever SWITCH receives a new input message having SOUTPUT's terminal as a destination. On receipt of a new message command, (which returns the header), SOUTPUT outputs the header (appropriately reformatted) and then outputs each block of the message. Blocks are read from MEMORY; they are printed by calling SUBSCRIBER.write for each character. Because new input of higher precedence may be inserted in LINEOUTPUT (by SWITCH) while SOUTPUT is writing out a message, SOUTPUT needs to know when a preemption is to occur. LINEOUTPUT communicates with SOUTPUT by setting a pre-empt flag. SOUTPUT periodically checks the flag and, if set, exits the loop. This results in the new, higher precedence message being received from LINEOUTPUT. As coded in Figure 5.13, SOUTPUT checks the pre-empt flag after each block of output. This could readily be changed to character level pre-emption by moving the check (when statement) inside the inner repeat statement.
The second kind of output command is acknowledge input. This is sent by SWITCH to inform the user at a terminal that an input message has been received. The action of SOUTPUT is to send the set of characters "input X received" to the SUBSCRIBER where X is the sequence number from the input message header.

The cancel kind of output comes about when a normal (new message) kind of output is cancelled. In this case, SOUTPUT does not output the message but merely informs the terminal user that output was cancelled. Because of the pre-emption mechanism, output in progress can be cancelled.

The stop input command is used to tell the terminal user to stop sending input. (This command is issued by MEMORY or the operator). At some later time, the restart input command will be received by SOUTPUT who then tells the user to restart input.

The final type of command is used to temporarily stop output. SOUTPUT tells the user that output is being stopped and then waits for a REPLY (from SWITCH). On receipt of the REPLY, SOUTPUT resumes.

After processing any output command, SOUTPUT calls LINEOUTPUT.done and then loops back to receive the next command from LINEOUTPUT.receive.

The final component of each subscriber group is a SUBSCRIBER device module. It provides an interface to one terminal by defining read and write operations called by SINPUT and SOUTPUT, respectively. To effect IO, SUBSCRIBER contains two character buffers, one for input and one for
Figure 5.13
Subscriber Output Process

process SOUTPUT;
    use header, block, LINEOUTPUT, MEMORY, SUBSCRIBER, NOTICE, REPLY;
    const line = # of output line i;
    var kind : integer; (*kind of output message*)
        hd : header; (*header of output message*)
        nblocks : integer (*number of blocks to output*)
        cblocks : integer (*number of blocks currently being output*)
        bl : block (*block of data to output*)
        msgcount: integer (*count of messages output*)
        i, j : integer (*local counters*)
    begin (*initialize variables*)
        msgcount := 0;
        LINLINEOUTPUT.receive (line,kind,hd);
    loop loop (*inner loop is for each output message; outer loop is to allow escape from inner loop when pre-emption occurs*)
        case kind of
            "new msg" : begin
                (*hd contains header of message to output*)
                nblocks := hd.size;
                inc(msgcount);
                (*format output header in bl*)
                (*output header contains: origin of message precedence, classification, identity local sequence number (msgcount)*)
                when hd.class is not valid for this terminal do NOTICE.post ("exception", invalid classification on line);
                LINLINEOUTPUT.done (line)
            exit;
        end;
Figure 5.13 (Continued)

Subscriber Output Process

bl := output header;
   i := 1;
repeat (*output, header on terminal*)
   SUBSCRIBER.write (bl[i]); inc(i)
until i > blocklength;
(*output contents of message as long as no
pre-emption occurs*)
j := 1;
repeat
   when pre-empt(line) do exit; (*go back to start
of main loop*)
   MEMORY.read (hd.filename, j, bl);
   i := 1;
   repeat (*output bl*)
      SUBSCRIBER.write (bl[i]);
      inc(i)
until i > blocklength;
   inc (j)
until j > nbloks;
end; (*of new msg case*)

"acknowledge input" : begin
(*tell terminal subscriber that an input message has
been received by SWITCH- hd contains the header of
the message*)
   bl := "input X received"; (*X is hd.sequence(i)*)
   i := 1
   repeat
      SUBSCRIBER.write (bl[i])
      inc(i)
until i > # of character in bl;
end (*of acknowledge input case*)
Figure 5.13 (Continued)

Subscriber Output Process

"cancel" : begin
  (*cancel output that was in progress - done
   automatically via pre-emption - here merely
   tell terminal operator that message was can-
   celled and then tell LINEOUTPUT that action
   is done*)
  bl := "output cancelled by supervisor";
  i := 1;
  repeat
    SUBSCRIBER.write (bl[i]);
    inc(i)
  until i > $# chars in bl;
  end (*of cancel case*)

"stopinput" : begin
  (*tell terminal operator to stop input*)
  bl := "stop input until told to restart";
  i := 1;
  repeat
    SUBSCRIBER.write (bl[i]);
    inc(i)
  until i > $# chars in bl;
  end (*of stop input case*)

"restart input" : begin
  (*tell terminal operator to restart input*)
  bl := "restart input from point where stopped"
  i := 1;
  repeat
    SUBSCRIBER.write (bl[i]);
    inc(i)
  until i > $# chars in bl;
  end (*of restart input case*)
Figure 5.13 (Continued)

Subscriber Output Process

"stop output" : begin (*stop writing output; wait for REPLY to signal proceed. Will restart output at beginning of stopped message*)
bl := "stopping output";
i := 1;
repeat
  SUBSCRIBER.write (bl[i]);
  inc(i)
until i > $ chars in bl;
REPLY.receive (line,kind); (*wait*)
end (*of stop output case*)
end;(*of case statement*)
LINEOUTPUT.done(line)
end end (*of loops*)
end SOUTPUT;
Subscribed Device

define module SUBSCRIBER;

define read, write;

var inr, outr, nrf : integer; (*input buffer vars*)
non r full, non r empty : signal; (*input signals*)
rbuf : array 1 : n of char; (*input buffer*)
inw, outw, nwf : integer; (*write vars*)
nonwfull, nonwempty : signal; (*output signals*)
wbuf : array 1 : n of char; (*output buffer*)

procedure read (var ch : char);

begin (*retrieve next input character from rbuf*)

if nrf = 0 then wait (nonempty) end;
ch := rbuf [outr];
outr := (outr mod n) + 1
de(nrf);
send (nonwfull);
end read;

procedure write (var ch : char);

begin (*deposit ch in output buffer*)

if nwf = n then wait (nonwfull) end;
wbuf[inw] := ch; inw := (inw mod n) + 1;
inc (nwf); send (nonwempty)
end write

process input;

(*input chars as long as rbuf is not full*)

begin

loop

if nrf = n then wait (nonrfull) end;
start read into buf[inr]; doIO;
inr := (inr mod n) + 1
inc (nrf);
send (nonempty);

end

end input;
Figure 5.14 (Continued)

Subscriber Device

process output;
(* output chars as long as wbuf is not empty *)

begin

loop

if nwf = 0 then wait (nonempty) end;

start output of buf[outw];
doIO;

outw := (outw mod n) + 1;

dec(nwf)

send(nonwfull);

end

end output

begin

inr := 1; outr := 1; nrf := 0; input;

inw := 1; outw := 1; nwf := 0; output

end SUBSCRIBER;
output. The procedures and processes synchronize with each other via counters, pointers and signals. Each buffer is treated as a circular queue where characters are deposited at one end and removed from the other. The SUBSCRIBER code is shown in Figure 5.14.

5.7 Trunk Groups

A trunk group provides an interface to a trunk line connecting one switching node to another. Each group has the same organization as a subscriber group (Figure 3.4). It contains TINPUT and TOUTPUT processes and a TRUNK device module. Code for these components is shown in Figures 5.16 - 5.18. As with subscribers, an actual system must contain one group for each trunk line.

The actions of each TINPUT and TOUTPUT process are basically the same as those of the SINPUT and SOUTPUT processes. The differences are that trunks transmit blocks instead of characters and that numerous control characters are used to control synchronization. Figure 5.15 defines a type for a trunkblock and also defines the kinds of control characters used for synchronization. Data blocks sent along trunk lines are 84 characters long. The control character at the start of a block is SOH or STX for the first and subsequent blocks of a message, respectively. The end character is ETX or ETB for the last and all previous blocks, respectively. Trunk blocks also contain a select character which defines the code used (e.g. ASCII), a parity, and 80
Figure 5.15
Trunk Blocks and Control Characters

type trunkblock
    record control, select : char;
    data : block; (*block = array of thars*)
    end, parity : char end;

Control Characters to Synchronize Transmission:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK1, ACK2</td>
<td>acknowledge last block; alternate ACK1, ACK2, ACK1, ACK2, etc.</td>
</tr>
<tr>
<td>NAK</td>
<td>non-acknowledge of block</td>
</tr>
<tr>
<td>STOP</td>
<td>stop transmission</td>
</tr>
<tr>
<td>RESTART</td>
<td>restart transmission</td>
</tr>
<tr>
<td>SYN</td>
<td>synchronize - used to keep line active when no data is being transmitted</td>
</tr>
<tr>
<td>CAN</td>
<td>cancel transmission</td>
</tr>
<tr>
<td>INV</td>
<td>invalid transmission</td>
</tr>
<tr>
<td>REP</td>
<td>repeat last block</td>
</tr>
<tr>
<td>RM</td>
<td>error - unable to frame block</td>
</tr>
</tbody>
</table>

Control Characters to Frame Blocks:

<table>
<thead>
<tr>
<th>Kind</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>start of header</td>
</tr>
<tr>
<td>STX</td>
<td>start of non header block</td>
</tr>
<tr>
<td>ETB</td>
<td>end of block but not end of message</td>
</tr>
<tr>
<td>ETX</td>
<td>end of message</td>
</tr>
</tbody>
</table>
characters of data.

For each data block transmitted, at least one control character is returned. Normally this acknowledges receipt of the block (ACK 1 or ACK 2). Exceptions can occur, however, and are indicated by the other control characters. In order to understand this message protocol in detail, the code for TINPUT and TOUTPUT should be studied carefully.

The TRUNK device module provides five operations: read, write, write control, post control and wait control. Read and write are used to transmit data blocks. The other operations are used to transmit and synchronize control characters. Write control is used to output a control character; it is called by TINPUT to respond to an input block. Waitcontrol is called by TOUTPUT to wait for a response from the prior output of a data block. The response is sent by the switching node at the other end of the trunk line and consequently, is received as input by TINPUT. Since TINPUT and TOUTPUT are processes, they can only communicate via an interface module. Therefore TINPUT passes control characters to TOUTPUT by calling the post control operation of TRUNK.

We now turn to the code of the processes. TINPUT (Figure 5.16) receives a trunk block, processes the information in the block and then loops. The TRUNK device module determines the type of input in each trunk block. There are three input types handled by TINPUT: error, control, and message.

If an input error occurs, meaning that TRUNK read an
invalid first character, TINPUT writes an RM (unable to frame) control character on the output line of the trunk by calling TRUNK.writecontrol. The switching node at the other end of the trunk will then send a cancel character to TINPUT. (see the code for TOUTPUT, Figure 5.17, since the output process at the other end of the trunk is in fact a TOUTPUT process of the other switching node).

If TINPUT reads a control character, it looks at the character and then takes an appropriate action. Characters which are responses from output are sent to TOUTPUT by calling TRUNK.writecontrol. A SYN character merely keeps the line "alive" so TINPUT does nothing. A CAN (cancel) character causes TINPUT to cancel input processing and notify SWITCH. An INV (invalid) character indicates problems so SWITCH is notified of an exception. Finally a REP (repeat) character should not occur so will be ignored.

The third type of input is message. This means that the TRUNK device module read a data block beginning with an SOM or STX control character. In this case, TINPUT processes the block in the same manner as SINPUT: header blocks are parsed and sent to LINEINPUT; a file is created for new messages; and blocks are written on MEMORY and the ACTIVE-ARCHIVE. When the last block of a message is read, SWITCH is notified.

TOUTPUT has the same organization as SOUTPUT (Section 5.6, Figure 5.13). Its program is given in Figure 5.17. TOUTPUT receives an output command from LINEOUTPUT and processes the command. There are five types of commands (the same ones as
process TINPUT; (*one copy per trunk*)

use block, header, trunkblock, TRUNK, NOTICE, REPLY, MEMORY, LINEINPUT, ACTIVE-ARCHIVE, HEADERS;

var bl : block; (*data block for MEMORY*)

head : header; (*header for input message*)
tbl : trunkblock; (*input block from TRUNK*)
cchar : char; (*input control character*)
type : integer; (*type of input from TRUNK*)
status : integer; (*status of msg - find start or in body*)
name : integer; (*internal message id*)
filename : integer; (*id of memory file*)
nblocks : integer; (*number of blocks in msg*)
ackno : integer; (*1 or 2 for ACK1 or ACK2*)
msgid : integer; (*external id of message*)

begin (*initialize variables*)
status := "find start";
ackno := 1;

loop TRUNK.read (tbl, intype); (*get next input*)

case intype of
error : (*invalid first character in input - unable to frame input*)

begin TRUNK.writecontrol("RM") end;

control : (*input is a control character*)

begin

ccchar := tbl.control; (*fetch control character*)

if cchar = "ACK1" or char = "ACK2" or char = "NAK" or char = "STOP" or char = "RESTART"

then (*give TOUTPUT the control character*)

TRUNK.postcontrol (ccchar)

elseif cchar = "SYN"

then (*do nothing - merely line synchronization character so just do next read*)
Figure 5.16 (Continued)

Trunk Input Process

```plaintext
elseif cchar = "CAN"
  then (*cancel input if in msg - otherwise ignore*)
    if status = "inbody"
      then (*tell SWITCH to cancel*)
        NOTICE.post ("input cancel", name);
        status := "find start";
        acknO := 1; end
    end
elseif cchar = "INV"
  then (*unsolicited answer*)
    NOTICE.post ("exception", number)
elseif cchar = "REP"
  then (*repeat character - will ignore for now -
       will assume acknowledgement has been sent*)
else (*error*)
  TRUNK.writecontrol ("RM")
end
end (*of control case*)

msg : (*input is a block of a message*)
 (*it starts with an SOM or STX control character*)
begin
  check parity of trunkblock tbl and check control characters
  if input is in error
    then TRUNK.writecontrol ("NAK")
  else (*acknowledge receipt of input*)
    if acknO = 1 then TRUNK.writecontrol ("ACK1");
    acknO := 2
    else writecontrol ("ACK2"); acknO := 1 end;
end
```
Figure 5.16 (Continued)

Trunk Input Process

if tbl.control = "SOH"
    then (*start of message header*)
        parse contents of tbl to build head;
        LINEINPUT.sendhead (head);
        REPLY.receive (trunk line $, name);
        (*name is internal msg id*)
        (*create an auxiliary storage file*)
        MEMORY.create (head,identity, hd.size, .filename);
        nblocks := 0;
        status := "in body"
        ACTIVE-ARCHIVE.data (hd.identity, tbl.data);
    elseif tbl.control = "STX" and tbl.end = "ETB"
        then (*block of message - not end*)
            bl := tbl.data; (*message itself*)
            MEMORY.write (filename,bl);
            ACTIVE-ARCHIVE.data (msgid,bl);
            inc(nblocks)
        elseif tbl.control = "STX" and tbl.end = "ETX"
            then (*end of message*)
                bl := tbl.data;
                MEMORY.write (filename, bl);
                MEMORY.endwrite (filename);
                inc(nblocks);
                ACTIVE-ARCHIVE.data (msgid,bl);
                (*store file size and filename in header*)
                HEADERS.retrieve (hd, name);
                hd.size := nblocks;
                hd.filename := filename;
                HEADERS.update (hd, name);
                (*notify SWITCH and reinitialize*)
                NOTICE.post ("end of input", name);
                status := "find start"
                ackno := 1;
            end (*of conditional*)
        end (*of msg case*)
    end (*of conditional*)
end (*of loop*)
end TINPUT;
for SOUTPUT): new message, cancel, stop input, restart input, and stop output.

On receipt of a new message command, TOUTPUT fetches and outputs each block of the message. The header is first re-formatted. Then TOUTPUT repeatedly writes a trunk block, waits for a control character response (actually read by TINPUT), checks for pre-emption and processes the control character. Iteration continues until either output is pre-empted, all blocks are written or an error occurs. If the control character correctly acknowledges the previous write then the next block (if any) is read from MEMORY. If the previous TRUNK write is not acknowledged, output is re-tried (up to some number, n, times). If the control character indicates that the output was in error then output is cancelled and SWITCH is notified.

The cancel, stop input, or restart input commands respectively cause a CANCEL, STOP, or RESTART control character to be written on the output line. The stop output command causes TOUTPUT to wait for a REPLY.

After any command is processed, TOUTPUT calls LINEOUTPUT to say that it is done. TOUTPUT then receives the next output command.

The final trunk component is the TRUNK device module. As mentioned before, it defines five operations: read, write, write control, post control, and wait control. Actual output is carried out by driver processes named input and output. Input fills a trunk block buffer for read. Output empties
Figure 5.17
Trunk Output Process

process TOUTPUT; (*one copy per trunk*)

use header, trunkblock, block, LINEOUTPUT, MEMORY, NOTICE,
REPLY, TRUNK;

const line = # of output line for trunk;

var kind : integer; (*kind of output message*)
head : integer; (*header for output message*)
nblocks : integer; (*number of blocks to output*)
cblock : integer; (*number of block currently being output*)

tbl : trunkblock; (*output to TRUNK*)
bl : block; (*data from MEMORY*)
cchar : char; (*control character*)
acknO : integer; (*1 or 2 for ACK1 or ACK2*)
more : Boolean; (*control for output loop*)
NAKtries : integer; (*number of tries at retransmission*)
REPtries : integer; (*number of waits for response*)

begin (*initialize variables*)

NAKtries := 0;
REPtries := 0

loop loop (*inner loop executed once per LINEOUT msg*)
(*outer loop allows escape on preemption*)

LINEOUTPUT.receive (line, kind, head);

case kind of

new msg : begin (*head contains header of message to output*)
nblocks := hd.size; (*no. of blocks to output*)

(*format output header in tbl.data - origin of message,
identity, precedence, classification sequence numbers*)
tbl.data := header contents as above;
tbl.control := "SOH";
tbl.end := "ETB";
tbl.parity := block parity;
more := true; acknO := 1; cblock := 0;
Figure 5.17 (Continued)

Trunk Input Process

(*main output loop - executed once for each message block*)

while more do
  TRUNK.write (tbl);
  TRUNK.wait control (cchar);
  (*check pre-emption*)
  when pre-empt [line] do
    TRUNK.writecontrol ("CAN") (*cancel*)
    exit;
  (*control may say to STOP; if so wait for restart*)
  if cchar = "STOP" then while cchar ≠ "RESTART" do
    TRUNK.waitcontrol (cchar) end
  end;
  (*take action depending on value of cchar*)
  if (cchar = "ACK1" and ackn0 = 1) or
    (cchar = "ACK2" and ackn0 = 2) then (*valid response*)
    inc (cblock);
    if cblock > nblocks then more := false (*output complete*)
  else (*get next block and build output block*)
    MEMORY.read (head.filename, cblock, bl);
    tbl.control := "STX"
    if cblock = nblock
      then tbl.end := "ETX"
      else tbl.end := "ETB" end;
    tbl.data := bl;
    tbl.parity := block parity;
    if ackn0 = 1 then ackn0 := 2
    else ackn0 := 1 end
  end
elseif cchar = "NAK" then (*do nothing - will retransmit same block - but
  if done more than n times notify supervisor
  and get next output*)
  inc (NAKtries);
Figure 5.17 (Continued)

Trunk Output Process

if NAKtries > n then NOTICE.post ("exception", $) 
NAKtries := 0; 
more := false (*stop trying - go to next output*) 
end
elseif cchar = "RM" 
then (*unable to frame - send cancel and notify local supervisor*) 
TRUNK.writecontrol("CAN"); 
NOTICE.post("exception", no); 
more := false; (*get next output*)
else (*cchar = "NONE" or is invalid*) 
(*TRUNK got no response in expected time - send REP to other trunk*) 
inc(REPtries); 
if REPtries < 8 
then (*ask again*) 
Trunk1.writecontrol ("REP") 
else (*cancel and tell supervisor*) 
Trunk.writecontrol ("CAN"); 
NOTICE.post ("exception", no); 
more := false; 
REPtries := 0
end (*of conditional statement*) 
end (*of while loop*)
(*end of message output*) 
end; (*of new msg case*)
cancel : (*cancel output in progress - requested by supervisor*) 
begin 
TRUNK.writecontrol ("CAN"); 
end; (*of cancel case*)
Figure 5.17 (Continued)
Trunk Output Process

stop input : ("tell output process on other end of the trunk to stop sending input")

begin
  TRUNK.writecontrol ("STOP")
end; (*of stop input case*)

restart input : ("tell output process on other end of trunk to restart sending input")

begin
  TRUNK.writecontrol ("RESTART")
end; (*of restart input case*)

stop output: ("temporarily stop sending output - signalled by the supervisor")

begin
  (*wait for reply signal to proceed*)
  REPLY.receive (line, kind)
end; (*of stop output case*)
end; (*of case statement*)
LINEOUTPUT.done (line) (*tell LINEOUTPUT that output message has been processed*)
end end (*of loops*)
end TOUTPUT;
either the control character buffer filled by writecontrol or a
trunk block buffer filled by write. Single buffers are used
so input and output synchronize with the operations via Boolean
and signal variables. The code of each part of TRUNK is
straightforward and is shown in Figure 5.18.

An interesting aspect of trunks is the timing of
physical input and output. Input is received one character
at a time until either a control character or entire block
has been read. Similarly output puts one character at a
time on the trunk's output line. We assume that trunks do
not give interrupts but instead provide or expect characters
to be periodically input or output. Therefore Modula's
doIO statement is not used. Instead, the drivers synchronize
by waiting for trunkticks which are periodically supplied by
the CLOCK process (Figure 5.3). This is quite different from
the interrupt drive IO used in a SUBSCRIBER (Figure 5.14).

5.8 Switch Process

The SWITCH process controls all activity in the switching
node. It accepts new input from input control processes, gen-
erates output commands, communicates with the operator, and
handles all exceptions. Its interface to other modules was
shown in Figure 3.8 and its actions were summarized in Figure 4.2.
Its program is listed in Figure 5.19.

All communication to SWITCH is via the NOTICE interface
module. SWITCH repeatedly receives a notice and processes
it. At the start of Figure 5.19, a large comment outlines
Figure 5.18
Trunk Device

device module TRUNK; (*one per trunk line*)
define  read, write, writecontrol, postcontrol, waitcontrol;
use trunkblock, trunktick;

var  inbuf, outbuf : trunkblock;
     intype : integer;
     outcchar, postchar : char;
     infull, outfull, outcfull, postfull : Boolean;

     doread, readdone : signal;
     dopost, postavail : signal;
     outbufempty, outccharempty : signal;

procedure read (var tbl: trunkblock; var kind : integer);
begin (*get next input from trunk line*)
    if not infull then wait (readdone) end;
    tbl := inbuf;
    kind := intype; (*control, msg, or error*)
    infull := false;
    send (dorread)
    end read;

procedure write (tbl : trunkblock);
begin (*fill buffer - output process will test outfull*)
    if outfull then wait (outbufempty) end;
    outbuf := tbl;
    outfull := true
    end write;

procedure writecontrol (c : char);
begin (*fill control character buffer*)
    if outcfull then wait (outcchar empty) end;
    outcchar := c;
    outcfull := true
    end writecontrol;
Figure 5.18 (Continued)

Trunk Device

procedure postcontrol (c : char);
begin (*fill post character buffer*)
  if post full then wait (dopost) end;
  postchar := c;
  postfull := true;
  send (postavail)
end postcontrol;

procedure waitcontrol (var c : char);
begin (*get posted character when available*)
  if not postfull then wait (postavail) end;
  c := postchar;
  postfull := false;
  send (dopost)
end waitcontrol;

process input;
(*do trunk input into inbuf*)
var : nch : integer; (*no. of character read*)
begin
  loop
  if infull then wait (doread) end;
  (*get first character*)
  wait (trunktick); (*clock synchronization*)
  inbuf.control := first character on line;

  if inbuf.control has even parity
  then (*control character*)
  intype := "control"
  elseif inbuf.control = "SOH" or
  inbuf.control = "STX"
  then (*message block*)
  intype := "msg"
  else intype := "error"
Figure 5.18 (Continued)

Trunk Device

if intype = "msg" then (*input block*)
    wait (trunktick);
    inbuf.select := next char;
    nch := 1;
    repeat wait(trunktick);
        inbuf.data[nch] := next char;
        inc(nch)
    until nch > blocklength;
    wait(trunktick);
    inbuf.end := next char;
    wait (trunktick);
    inbuf.parity := next ohar
end (*of block input*)
infull := false;
send (readdone)
end (*of loop*)
end input;

process output
(*output control characters from outcchar or blocks from outbuf or, if both are empty, output a line synchronization signal*)
var
    nch : integer; (*character count*)
begin
    loop
        wait (trunktick); (*clock synchronization*)
        if outcfull then (*output control character*)
            put contents of outcchar on line;
            outcfull := false;
            send (outcchar empty)
        elseif outfull then (*output block*)
            put outbuf.control on line;
            wait (trunktick);
            put outbuf.select on line;
            nch := 1;

        else
            wait (trunktick);
            put outbuf.control on line;
            wait (trunktick);
            put outbuf.select on line;
            nch := 1;
end loop
end output;
Figure 5.18 (Continued)
Trunk Device

repeat wait (trunktick);
    put outbuf.data [nch] on line;
    inc(nch)
until nch > blocklength;
wait (trunktick);
put outbuf.end on line;
wait (trunktick);
put outbuf.parity on line;

outfull := false;
end (outbufempty)
else (*output SYN character - there is no real output
so just keep line synchronized*)
    put "SYN" on line
end (*of conditional*)
end (*of loop*)
end output;

begin (*initialize TRUNK*)
    unfull := false; outfull := false; outcfull := false;
    postfull := false;
    input; output
end TRUNK;
the program and enumerates the kinds of notices. The actions
SWITCH takes for each notice will now be discussed in the
order in which they appear in the program.

A "head" notice, sent by LINEINPUT, signals the presence
of a new header. SWITCH receives the header from LINEINPUT and
records control information for the new message. The header
is entered into HEADERS, a name table entry is constructed,
and an action message is sent to ACTIVE-ARCHIVE and the oper-
ator. Finally a REPLY is given to the input control process
which input the header.

The largest case SWITCH handles occurs when an input
control process sends an "end of input" notice. First,
the header is retrieved from HEADERS. Second, the "end of
input" action is recorded on the archive and sent to the
operator. Third, for input from local subscribers, an ack-
nowledgement is sent to the subscriber via LINEOUTPUT. Fourth,
potential message orbit is checked for. An orbit occurs if
a message ever comes back to a switching node which has pre-
viously processed it. This could happen if directory entries
(see below) are erroneous or if trunk lines go down so two
switching nodes use each other as alternate routes to a third
node. An orbit is detected as follows. When a message is
output, SWITCH stores the local switching nodes' identity in
the sequence array of the message header. On input, SWITCH
looks to see if its identity is already in sequence. If so
an orbit has occurred and the operator is informed (he will
most likely cancel the message). The fifth action SWITCH
takes on "end of input" notices is to format output commands. For each destination, the directory is consulted to find the first line with "ok" status. If one is found, the header is inserted in LINEOUTPUT. If none is found the operator is informed. After all destinations have been processed, the header is put back in HEADERS for future reference.

Once output completes at any destination, LINEOUTPUT posts a "done" notice. SWITCH decrements the output count (number of destinations) of the message. If the count becomes zero, SWITCH then archives an "output complete" action, tells the operator, deletes the header from HEADERS and destroys the message's MEMORY file.

The above three kinds of notices ("head", "end of input", and "done") pertain to normal message processing. The other kinds should occur much less frequently since they pertain to exceptions and operator requests.

The "stop" notice is issued by the operator or MEMORY to stop input or output. SWITCH sends an output command to the output controller for the line. To restart IO, a "restart" notice is sent to SWITCH. Input is restarted by sending an output command to the subscriber or trunk. Output is restarted by giving a REPLY.

In order to allow the operator to monitor system status, SWITCH also accepts a "status" kind of notice. Its actions are to retrieve the appropriate status value(s), format a message, and send it to the operator. Details for each type of status which might be useful are left unspecified here.
If an input control process finds the cancel sequence of characters in an input message, it notifies SWITCH.

SWITCH cancels a message by deleting its header and destroying its file. As usual, the archive and operator are informed.

Output can be cancelled at the request of the operator. To cancel a message, first its internal name is looked up in the names table. If it is not found the operator is informed. If it is found, the message's header is retrieved. If the message has previously been sent to LINEOUTPUT, LINEOUTPUT.cancel is called. (The message may not have been sent because an orbit might have occurred). Cancellation is then archived, the header deleted, and the file destroyed.

The next case processes "exception" notices. The types of exceptions currently implemented deal with archive tapes. Others would also exist in an actual implementation. For each type of exception, a message is formatted and sent to the operator.

"Alter" notices are sent by the operator to alter the contents of either the directory or the line status table. SWITCH receives the new values from SUPERVISOR and stores them in the appropriate table entry.

The final kind of notice signals a pre-emption. When LINEOUTPUT sets a pre-empt flag (except on cancel), he notifies SWITCH who in turn tells the operator.

The complete listing of SWITCH follows as Figure 5.19.
process SWITCH;

use NOTICE, LINEINPUT, LINEOUTPUT, HEADERS, REPLY, ACTIVE-
ARCHIVE, SUPERVISOR, timeofday, header, actionmsg, operator-
output, operatorrequest

type name = record  (*controls for active messages*)
  msgid, intname : integer;
  outputcount : integer
  end;

destination = record  (*trunk or subscriber line no's*)
  primary, alt 1, alt 2 : integer
  end;

var names : array 1 : max$ activemsgs of name; (*of active mes-
sages*)

directory : array 1 : #destinations of destinations;
linestatus : array 1 : #lines of integer;

kind, data : integer;  (*input from NOTICE*)
hd : header;  (*local storage for input header*)
index : integer;  (*internal message name*)
actmsg : actionmsg;  (*output to archive*)
line : integer;  (*output line number*)
c,d,i : integer;  (*counters*)
orbit : Boolean;  (*message in loop*)
opout : operatoroutput;  (*output to SUPERVISOR*)
opreq : operatorrequest;  (*input from SUPERVISOR*)

body of switch is a loop with a case statement for
each kind of NOTICE SWITCH receives – the kinds of
notices are the following:

loop NOTICE.receive (kind, data)

case kind of

"head" :  data is 0  – new header from LINEINPL
"end of input" :  data is internalid  – end of input
"done" :  data is internalid  – end of output
"stop" :  data is line#  – stop a line
"restart" :  data is line#  – restart a line
"status" :  data is key for
  type of status  – send status to operator
"input cancel" :  data is internalid  – cancel message
"output cancel" :  data is externalid  – cancel message
Figure 5.19 (Continued)

Switch Process

"exception" :  data is key for
   type of exception - print message on operator's
                     console
"alter" :      data is 0       - get operator request to
   alter directory or line
"pre-emption" : data is msgid  - inform operator
                    status from SUPERVISOR

end

end

* * * * * * * * * * * * * * * * * * * *

begin  (*initialize tables*)
       (*details not shown*)

loop  NOTICE.receive (kind, data);

   case kind of

"head" :  begin  (*get new header*)
     LINEINPUT.receivehead (hd);
     hd.status := "in input";
     HEADERS.enter (hd,index);
     (*fill in name table*)

     with names(index) do
       msgid := hd.identity;  intname := index;
       count := hd.#destinations end;
     (*archive receipt of header*)

     with actmsg do
       msgid := hd.identity;  time := timeofday;
       action := "header received" end
     ACTIVE-ARCHIVE.action (actmsg);
     (*inform operator of action*)
     SUPERVISOR.sendoutput (actmsg);
     LINEOUTPUT.insert (operator, "output", o, low precedence);
     (*tell input to proceed*)
     REPLY.give (hd.origin, index)

end  (*of head case*)
"end of input" : begin

index := data; (*internal msg identifier*)
HEADERS.retrieve (hd, index);
(*archive end of input*)
with actmsg do

msgid := hd.identity; time := timeofday;
action := "end of input" end;
ACTIVE-ARCHIVE.action(actmsg);
SUPERVISOR.sendoutput (actmsg);
LINEOUTPUT.insert (operator, "output", 0, low);
(*acknowledge receipt of input if sender is a
local subscriber*)

if hd.origin is a subscriber

then LINEOUTPUT.insert (hd.origin, "acknowledge-
input", index, low) end;

(*check for message orbit*)

with hd do

i := 1; orbit := false;
while (i <= seqcount and not orbit)

do if sequence (i) = local switch#

then orbit := true end;

inc(i) end;

end

if orbit

then (*output message to operator*)

format orbit message
SUPERVISOR.sendoutput (operator output)
LINEOUTPUT.insert (operator, "operatoroutput",
0, low precedence)
hd.status := "orbit"; HEADERS.update (hd, index)

else (*proceed to output message to each destination*)

with hd do status := "output";
(*update sequence data*)
inc(seqcount); sequence (seqcount) := local switch!
(*output to each destination*)
c := names (index).outputcount; (*#dests*)
i := 1;
Figure 5.19 (Continued)

Switch Process

\[ \text{repeat} \]
\[ d := \text{destinations}(i); \]
\[ \text{line} := \text{directory}(d).\text{primary}; \]
\[ \text{if linestatus(line) \neq \text{"ok"}} \]
\[ \text{then line := directory}(d).\text{alt } 1 \]
\[ \text{if linestatus(line) \neq \text{"ok"}} \]
\[ \text{then line := directory}(d).\text{alt } 2 \]
\[ \text{if linestatus(line) \neq \text{ok}} \]
\[ \text{then format no good line message;} \]
\[ \text{SUPERVISOR.sendoutput (message);} \]
\[ \text{LINEOUTPUT.insert (operator, \text{"operator output"}, 0, low)} \]
\[ \text{end end end;} \]
\[ \text{if linestatus(line) = ok then} \]
\[ \text{LINEOUTPUT.insert(line, \text{"new msg"}, index, prec)} \]
\[ \text{hd.dests}(i) := \text{line;} \]
\[ \text{inc}(i) \]
\[ \text{until } i > c; \]
\[ (*\text{update header*}) \]
\[ \text{HEADERS.update (hd, index)} \]
\[ \text{end (*of end of input case*)} \]

\[ \text{"done" : begin (*output to one destination is complete*)} \]
\[ \text{index := data; (*identifies message*)} \]
\[ \text{dec(names(index)).outputcount);} \]
\[ (*\text{if output is all complete - archive completion and delete message from system*}) \]
\[ \text{if names(index).outputcount = 0} \]
\[ \text{then} \]
\[ \text{with actmsg do} \]
\[ \text{msgid := names(index).msgid; time := timeofday;} \]
\[ \text{action := \text{"output complete" end};} \]
Figure 5.19 (Continued)

Switch Process

ACTIVE-ARCHIVE.action (actmsg);
SUPERVISOR.sendoutput (actmsg);
LINEOUTPUT.insert (operator, "output", 0, low);
(*delete header and destroy memory file*)
HEADERS.delete (index);
MEMORY.destroy (names (index).msgid)
end
end (*of done case*)

"stop" : begin (*stop IO on one line - done by sending a message
to the output process associated with the line -
If the line is an input line, its output partner
will send a message to the human or other switch
node telling it to stop input*)
line := data; (*identifies line *)
if line is an input line
    then line := output partner's line no.
        LINEOUTPUT.insert (line, "stop-input", 0,
                         highest precedence)
    else LINEOUTPUT.insert (line, "stop-output", 0,
                         highest precedence)
end
end (*of stop case*)

"restart" : begin (*restart IO - technique is same as above*)
line := data;
if line is an input line
    then line := output partner's line no.
        LINEOUTPUT.insert (line, "start-input", 0,
                         highest precedence)
    else (*output process has gone to sleep because
         of stop message*)
        REPLY.give (line, 0)
end
end (*of restart case*)
"status" : begin  

(*data identifies type of status requested - possible types are
  1. status of line
  2. directory entry
  3. queue lengths, etc. *)

(*for the status - SWITCH formats a message*)
opout.data := contents of message;
opout.size := length of message;

(*send message to operator*)
SUPERVISOR.sendoutput (opout);
(*tell operator output process a message is in SUPERVISOR for him*)
LINEOUTPUT.insert (operator, "output", 0, high)

end (*of status case*)

"input cancel" : begin (*sent by an input process to cancel
  input of a message*)
index := data; (*identifies message*)

(*archive cancel action*,

with actmsg do

 msgid := names (index).msgid; time := timeofday;
  action := "cancelled input" end;
ACTIVE-ARCHIVE.action(actmsg);
opout.data := "cancelled input";
opout.size := 15;
SUPERVISOR.sendoutput (opout);
LINEOUTPUT.insert (operator, "output", 0, low);

(*delete header and destroy memory file*)
HEADERS.delete(index);
MEMORY.destroy(names (index).msgid)

end (*of input cancel case*)
"output cancel" : begin (*sent by operator to cancel output*)
   (*data gives external message identifier*)
   (*find internal name*)
   i := 1; index := 0;
repeat
   if names (i).msgid = data
      then index := names(i).internal id end;
      inc(i)
   until index ≠ 0 or i > max$activemessages;
   if index = 0
      then (*message not found*)
         opout.data := "message not found"
         opout.size := 17;
         SUPERVISOR.sendoutput (opout);
         LINEOUTPUT.insert (operator, "output", 0, high);
   else
      HEADERS.retrieve (hd, index);
      if hd.status = "output"
         then (*tell each output destination to cancel by telling LINEOUTPUT*)
            LINEOUTPUT.cancel (index) end;
      (*archive cancellation*)
      with act$msg do
        msgid.:= names (index).internal id;
         time := timeofday;
         action := "output cancelled" end
      ACTIVE-ARCHIVE.action (act$msg);
      opout.data := "output cancelled";
      opout.size := 15;
      SUPERVISOR.sendoutput (opout);
      LINEOUTPUT.insert (operator, "output", 0, low);
      (*delete header and destroy memory file*)
      HEADERS.delete (index);
      MEMORY.destroy (names(index).msgid)
      end (*of conditional*)
end (*of output cancel case*)
"exception": begin
 (*print exception message on operator's console*)
 (*data identifies exception type*)
 case date of
 "mount action tape": begin opout.data := "mount new action tape on
 active archive";
 opout.size := length of data end;
 "mount data tape": begin opout.data := "mount new data tape on
 active archive";
 opout.size := length of data end;
 "end action tape": begin opout.data := "end of action tape on old
 archive";
 opout.size := length of data end;
 "end data tape": begin opout.data := "end of data tape on
 old archive";
 opout.size := length of data end;
 end (*of case statement*)
 SUPERVISOR.sendoutput (opout);
 LINEOUTPUT.insert (operator, "output", 0, high)
 end (*of exception case*)

"alter": begin (*get operator request from SUPERVISOR
 alter either directory or line status*)
 SUPERVISOR.receiveq (opreq);
 with opreq do
 case key of
 "alter directory": (*values give line and new primary and alternate
 destinations*)
 begin with directory (value(1)) do
 primary := value (2); alt1 := value (3);
 alt2 := value (4) end
 end;
Figure 5.19 (Continued)

Switch Process

"alter line status" : (*values give line# and new status*)
begin linestatus(value(l)) := value (2) end
end (*of case*)
end (*of with*)
end (*of alter case*)

"preemption" : begin (*notify operator of pre-emption data is internal msgid*)
index := data
opout.data := names(index).msgid; (*external name*)
opout.size :=
SUPERVISOR.sendoutput (opout);
LINEOUTPUT.insert (operator, "output", 0, low)
end (*of pre-emption case*)
end (*of entire case statement*)
end (*of loop*)
end SWITCH;
5.9 Operator Group

The operator group consists of six components: input control process, output control process, SUBSCRIBER device module, SUPERVISOR module, RETRIEVE process, and old archive device module. The components are connected to each other as was shown in Figure 3.7. The switching node operator can send and receive messages in the same way as other subscribers. The operator can also make certain requests and receive control and exception output messages. Programs for each of the operator group components are presented in this section.

The operator is assumed to have an IO terminal which is identical with those for subscribers. Therefore the device interface in the operator group is a SUBSCRIBER device module identical to that in Figure 5.14.

The operator input control process is a slight modification of the SINPUT process of subscribers (Figure 5.12). The changes are shown in Figure 5.20; they result from the fact that the operator can generate two types of input: regular messages and operator requests. Regular messages are handled in a manner identical to that for subscribers. Operator requests are assumed to start with a distinguishing sequence of control characters. Their body consists of a key and up to four values. Once the start of an operator request is found (in the "find start" case), SINPUT's status is set to "find request." The request is then read
Figure 5.20
SINPUT Process for Operator

Make the following changes to SINPUT:

1. use: add SUPERVISOR, operatorrequest to use list
2. variables: add
   oreq : operatorrequest; (*kind and values of
   operatorrequest*)
3. "find start" case:
   add a search for start of operator request sequence,
   of control characters
   if start operator request found
     then current := 0;
     status := "find request" end;
4. add "find request" case as follows:
   "find request":
     begin (*build operator request*)
       if current = 0 then oreq.key := ch;
       inc(current)
       else oreq.value(current) := ch;
       inc(current)
     end
     if end of request then
case oreq.key of
  "status": begin
    NOTICE.post ("status", oreq.value (1))
  end;
  "cancel": begin
    NOTICE.post ("cancel", oreq.value (1))
  end;
  "wait": begin
    NOTICE.post ("wait", oreq.value (1))
  end;
"restart" : begin
  NOTICE.post ("restart", oreq.value (1))
end;

"alter" : begin
  SUPERVISOR.sendreq (oreq);
  NOTICE.post ("alter", 0)
end;

"new tape" : begin
  if oreq.value (1) = "active-archive"
    then ACTIVE-ARCHIVE.resume
    else OLD-ARCHIVE.newtape (oreq.value(1))
  end;

"retrieve" : begin
  SUPERVISOR.doretreive(oreq)
end;

"cancel retrieve" : begin
  SUPERVISOR.cancelretriev
derendend; (*of case statement*)

status := "find start" (*find next operator request
or start of message*)
end; (*of find request case*)
and stored one character at a time. Once finished, a case statement on the key is executed. Status, cancel, wait, and restart requests post a NOTICE for SWITCH. Alter sends the request data to SUPERVISOR and then posts a NOTICE (there are more data values to pass than post can accept). New tape requests signal that a new archive tape has been mounted so the appropriate archive (ACTIVE or OLD) is called. The retrieve request causes a retrieve message to be sent to the RETRIEVE process (see below) via SUPERVISOR. Finally, a retrieve can be cancelled by the cancel request. After processing an operator request, SINPUT for the operator sets status to "find start" to look for the next input.

The SOUTPUT process for the operator is only slightly changed from SOUTPUT processes in subscriber groups (Figure 5.13). The changes are shown in Figure 5.21. Since the operator receives special output commands, the change is to add one more case to process the one more kind of LINEOUTPUT. The data for the operator comes from SUPERVISOR. It is merely written out by calling SUBSCRIBER.write.

The SUPERVISOR module (Figure 5.22) interfaces the operator's SINPUT and SOUTPUT processes to SWITCH and RETRIEVE. It defines seven operations: sendreq, receivereq, sendoutput, receiveoutput, doretrieve, getretrieve, and cancelretrieve. Sendreq and receivereq are used to send operator requests from SINPUT to SWITCH. Available requests are stored within SUPERVISOR in a requests queue. Sendoutput and receiveoutput are used to send operator output messages from SWITCH to
Figure 5.21
SOUTPUT Process for Operator

Make the following changes to SOUTPUT:

1. **use** - add SUPERVISOR, operatoroutput
2. **variables** - add
   
   | opout : operatoroutput |

3. add "output" case as follows:

   "output" : begin
   (*fetch operator output data and write it out*)
   SUPERVISOR.receiveoutput (opout);
   i := 0;
   repeat
   SUBSCRIBER.write (opout.data[i])
   until i = opout.size
   end
SOUTPUT. They too are stored within SUPERVISOR in a queue. Doretrieve and getretrieve are used to send retrieve requests from SINPUT to RETRIEVE and are also queued within SUPERVISOR. Cancelretrieve is called by SINPUT to cancel a retrieve. It sets a flag which is exported from SUPERVISOR and examined periodically by RETRIEVE. This flag (stoppingretrieve) serves the same role as did the pre-empt flags in LINEOUTPUT. The code for each of the seven SUPERVISOR operations is straightforward. Note that SUPERVISOR could be broken into three separate interface modules since the operations work in pairs. We did not do so, however, because it makes sense to group all the SUPERVISOR interface operations together. In this way, the entire interface between the operator and other processes appears in one place.

The RETRIEVE process (Figure 5.23) processes retrieve requests sent by the operator input controller via SUPERVISOR. There are three kinds of retrieves: copy, retransmit, and trace. The kind of request is indicated by opreq.value[1] which is used as a case statement selector. The copy retrieve causes all blocks of a message (identified by opreq.value [2]) to be read from an old archive tape and printed on the operator's terminal. The retrieved message is sent to the operator as if it were a new message. Namely, RETRIEVE builds a header, sends it to SWITCH via LINEINPUT, and then retrieves each message block and stores it on a MEMORY file. When the end of the retrieved message is found, SWITCH is notified.

The retransmit type of retrieve causes an archived
interface module SUPERVISOR;

use operatorrequest, operatoroutput; (*data types*)

define sendreq, receivereq, sendoutput, receiveoutput, stopretrieve, doreretrieve, get retrieve, cancelretrieve

var requests : queue n of operatorrequest;
    requestavail, requests not full : signal;

output : queue n of operatoroutput;
    output not full : signal;

retrievals : queue n of operatorrequest;
    retrieval not full, retrieveavail : signal;

stopretrieve : Boolean; (*signals RETRIEVE process to stop a retrieval*)

procedure sendreq (opreq : operatorrequest);
    begin (*called by SINPUT*)

        if requests.size = n then wait (requests not full) end;
        requests.insert (opreq);
        send (requestavail)

    end

end sendreq;

procedure receivereq (opreq : operatorrequest);
    begin (*called by SWITCH*)

        if requests.empty then wait (requestavail) end;
        opreq := requests.remove;
        send (queue not full)

    end receivereq;

procedure sendoutput (opout : operatoroutput);
    begin (*called by SWITCH*)

        if output.size = n then wait (outputnotfull) end;
        output.insert (operatoroutput)

    end sendoutput;
Figure 5.22 (Continued)

Supervisor Module

procedure receiveoutput (opout : operatoroutput);
begin (*SOUTPUT knows message is available*)
output.remove (operatoroutput);
send (outputnotfull)
end receiveoutput;

procedure doretrieve (opreq : operatorrequest)
begin (*called by SINPUT*)
if retrievals.size = n then wait(retrievalnotfull)
end;
retrievals.insert (opreq);
send (retrieveavail)
end doretrieve;

procedure getretrieve (opreq : operatorrequest);
begin (*called by RETRIEVE*)
stopretrieve := false; (*it may have been on*)
if retrievals.empty then wait (retrieveavail)
retrievals.remove (opreq);
send (retrievalnotfull)
end getretrieve;

procedure cancelretrieve; (*called by SINPUT*)
begin stopretrieve := true
end cancelretrieve;

begin stopretrieve := false
end SUPERVISOR;
message to be fetched and sent just as if it were a new message. The message is processed in the same way as for copy above except that the header used is the message's original header. For the retransmit case RETRIEVE acts exactly like an SINPUT process.

The final type of retrieval is the trace which causes all actions taken on a message to be printed on the operator's console. This is accomplished within RETRIEVE by building a header to direct a message to the operator, reading actions from OLD-ARCHIVE and storing each action as a block on MEMORY. The only difference between copy and trace is that the former retrieves the data in a message while the latter retrieves actions taken on a message.

To retrieve data or actions from archived messages, RETRIEVE calls the OLD-ARCHIVE device module (Figure 5.24). OLD-ARCHIVE defines three operations (retrieveaction, retrievedata, and newtape) and contains two driver processes (datatape and actiontape). Retrieveaction is called to retrieve the next action with a given id from the currently mounted action tape (a different action tape than the one being filled by ACTIVE-ARCHIVE). It does so by searching tape blocks for an action message with the appropriate id. Once an action message is found it is returned. When retrieve action requires the next input block on the action tape, it signals the actiontape driver and waits. If the end of the action tape is reached, a notice is sent to SWITCH which in turn informs the operator. Once the operator
process RETRIEVE;

use SUPERVISOR, stopretrieve, OLD-ARCHIVE, operatorrequest, operatoroutput, header, block, actionmessage, msgid, REPLY

var oreq : operatorrequest;
opout : operatoroutput;
hd : header;
bl : block;
act : actionmessage;
id : msgid; result : integer; cnt : integer;

begin loop (*main loop - once per retrieval*)
SUPERVISOR.getretrieve (oreq);

(*in oreq, value[1] names the type of request to
perform - they are: copy - print message on
operator's console
retransmit - re-send message to
destination
trace - print all actions taken
on message on operator
console
value [2] identifies the message to retrieve*)

id := oreq.value [2];
case oreq.value [1] of

"copy" : begin (*copy entire message on operator's console*)
build new header in hd - treat retrieved message
as new input sent to operator;
put copy of header in bl;
LINEINPUT.sendhead (hd); REPLY.receive ("line$", MEMORY.create (id, size); MEMORY.write (id, bl);
loop (*get body of message*)
OLD-ARCHIVE.retrievedata (id, bl);
MEMORY.write (id, bl); (*put block back on auxiliary
memory*)
when end of msg in bl or stop retrieve do exit;
end (*once end is found signal SWITCH*)
NOTICE.post ("end", id)

end (*of copy case*)
"retransmit": begin
(*this case is like copy except actual header is used to direct the output*)
OLD-ARCHIVE.retrievedata (id, bl);
transfer items from bl to hd;
LINEINPUT.sendhead (hd);
REPLY.receive ("line", result);
MEMORY.create (id, size);
MEMORY.write (id, bl);
loop
OLD-ARCHIVE.retrievedata (id, bl);
MEMORY.write (id, bl);
when end of msg is in bl or stop retrieve do exit
end
NOTICE.post ("end", id)
end (*of retransmit case*)

"trace": begin
(*retrieve all actions taken on a message and print them on the operator's console - build message out of actions and route it as for normal messages*)
build header - destination is operator's console
LINEINPUT.sendhead (id);
REPLY.receive ("line", result);
MEMORY.create (id, size);
loop (*retrieve actions*)
OLD-ARCHIVE.retrieveaction (id, act);
copy act into bl;
MEMORY.write (id, bl);
when last action on msg or stop retrieve do exit
end
NOTICE.post ("end", id)
end (*of trace case*)
end (*of main loop*)
end RETRIEVE;
has mounted a new action tape, he inputs a new tape command which causes the operator's SINPUT process to call OLD-ARCHIVE.newtape. This reinitializes OLD-ARCHIVE variables and causes the first tape block to be read. Once reading is complete, retrieve action is signalled and proceeds as above.

With this scheme, the operator must maintain a catalogue indicating which messages are on which archive tapes. It is his responsibility to mount the appropriate tape before issuing a retrieve request. OLD-ARCHIVE merely reads from where he is on the currently mounted tape and informs the operator when the end of tape is reached. The operator must then mount the appropriate next tape.

The retrieved data operation is called by RETRIEVE to fetch the next message block with a given id from the currently mounted data tape. Its implementation is analogous to that for retrieve action.

The two driver processes in OLD-ARCHIVE read the next record from the data and action tapes when signalled to do so. Their implementation is straightforward. The entire program for OLD-ARCHIVE follows as Figure 5.24.

5.10 System Initialization

The code for each process and module of the message switch system has now been described. All that remains to complete the switching node program is code to initialize the main processes. The initialization code is shown in Figure 5.25.
-122-

Figure 5.24
Old Archive Device

device module OLD-ARCHIVE;

define retrieveaction, retrievedata, newtape;

use block, actionmsg

const actiontapesize = m_1; (*same values*)
datatapesize = m_2; (*as in ACTIVE-ARCHIVE*)
actionrecordsize = n_1;
datrecordsize = n_2;

var arnO, drnO, abnO, dbnO : integer;
    (*current count of records and blocks*)
actionbuffer : array 1 : actionrecordsize of
    actionmsg; (*record from action tape*)
datbuffer : array 1 : datatrecordsize of record id :
    integer; info : block end; (*record from data tape*)
inputaction, actiendone, inputdata, datadone : signal;
actionavail, dataavail : Boolean;
actionstatus, datastatus : integer;
tapemounted : signal; (*end of tape synch.*)

procedure retrieveaction (id : integer; var msg : actionmsg);
    (*retrieve next actionmsg with identifier id*)
begin
    loop (*loop until find action*)
        inc(abnO);
        while abnO <= actionrecordsize do (*look at actions*)
            when actionbuffer (abnO).msgid = id (*got it*)
                do msg := actionbuffer (abnO);
                result = "found it"
                exit
                inc(abnO)
        end;
end;
Figure 5.24 (Continued)
Old Archive Device

(*end of record so fetch next one if possible*)
if arno = actiontapesize
  then NOTICE.post ("exception", "end of action tape"
    wait (tapemounted) end
  actionavail := true; signal (inputaction)
  wait (actiondone);
  inc (arno); abn0 := 0 (*next record, first block*)
end (*of loop*)
end retrieve action;

procedure retrievedata (id : integer; var bl : block);
(*retrieve next block with identifier id*)
begin
  loop (*until find data*)
    inc(dbno) (*look in current record*)
    while dbno < datarecordsize do
      when databuffer (dbno).id = id
        do bl := databuffer (dbno).info;
          result := "found it" exit
        inc (dbno)
      end
    (*end of record so fetch next one if possible*)
    if drno = datatapesize
      then NOTICE.post ("exception", "end of data tape")
        wait (tapemounted) end;
      dataavail := true; signal (inputdata);
      wait (datadone);
      inc(drno); dbno := 0 (*next record, first block*)
    end (*of loop*)
end retrievedata;
Figure 5.24 (Continued)

Old Archive Device

procedure newtape (kind : integer);
begin (*initialize and get first record from new tape*)
if kind = action
then arno := 0; abno := 0;
   actionavail := true; signal (inputaction);
   wait (actiondone);
   inc (arno)
else drno := 0; dbno := 0;
   dataavail := true; signal (inputdata);
   wait (datadone);
   inc (drno)
end
signal(tapemounted)
end newtape

process datatape;
begin loop
if not dataavail then wait (inputdata) end;
initiate read into databuffer
doIO
if error then datastatus := "error"
else datastatus := 0 end;
dataavail := false; signal (datadone)
end
data tape;

process actiontape;
begin loop
if not actionavail then wait (input action ) end;
initiate read into action buffer; doIO;
if error then actionstatus := "error"
else actionstatus := 0 end;
actionavail := false; signal (actiondone)
end
end actiontape
Figure 5.24 (Continued)

Old Archive Device

("initialize device module")
begin
arnO := 0; abnO := 0; drnO := 0; dbnO := 0;
dataavail := false; actionavail := false;
datatape; actiontape
end

OLD-ARCHIVE
-126-

Figure 5.25
System Initialization

\textbf{begin} (*activate each main process*)

for each subscriber group and the operator do

- \textbf{SINPUT;}
- \textbf{SOUTPUT;}
- \textbf{SWITCH;}
- \textbf{RETRIEVE}

\textbf{end}
6.0 Summary and Evaluation

The description of a representative message switching communications system and its implementation in Modula have now been completed. This chapter summarizes the presentation and discusses its relation to the design process. The utility of Modula as a design language is then evaluated.

6.1 Summary of Design Technique

The design presented here has been described in the same order in which it was developed. It evolved from a sequence of five steps. First, the basic system functions were identified and described (in Chapter 2). System functions in this case were specified by communications people who use message switching systems. My role at this stage, as the designer, was to discuss the functions with intended users so that we could both come to agreement on the purpose and scope of the proposed system and become comfortable with each other's vocabulary. In addition, it was (and generally is) helpful to lay out a typical hardware configuration in order to get a better feel for the size and nature of the system. The hardware need not be considered in detail at this point, however.

The second step was to specify in detail the formats of input and output messages. (This was also done in Chapter 2). System functions describe how information is processed; this step in the design describes what is processed. It completely
characterizes the user's view of the system.

The third step was to develop an organization, in terms of Modula constructs, for a system having the functions enumerated in step one. Modula has processes and modules as its basic building blocks for multiprogramming systems. Various organizations were considered at this stage, all in terms of how the functions could be realized using processes and modules. The organization settled on was shown in Figure 3.3. Refinements of the groups in Figure 3.3 were shown in Figures 3.4 - 3.8. The actual design proceeded in exactly this manner. First important groups of processes and modules were identified; in this case the groups implement IO interfaces and the central switching function. In general, a similar correspondence of groups to IO devices and major system functions should occur. Each group was in turn refined into modules and processes. Finally, the interconnection of groups, in this case the interface between IO groups and the SWITCH process, was organized in terms of modules.

The fourth step was to list the actions of each major system component. This was presented in Chapter 4. In the message switch there are four major components: input, switch, output, and the operator. The first three correspond to the phases which an individual message goes through as it is processed by the system. For each of these components, a path description of its actions was developed. Once this was complete, an ordered list of the actions taken in pro-
cessing each message was compiled. (Figure 4.4). At this point, the design was discussed in detail with the people who had contracted for the work. This allowed misconceptions and ambiguities about the system functions to be clarified. It helped me to be sure of the direction I was headed and helped the contractors to better understand the program they would be getting.

The fifth step was to actually program each component. No programming was done (or should never be done) until the organization and system actions were understood by myself, and agreed to by the contractor. Once the organization of the whole system is well understood, it becomes relatively easy to program each component. This is not to say that creativity is no longer needed though! Programming large components (e.g. MEMORY or even LINEOUTPUT) is still a rewarding challenge.

These five steps - system functions, IO interfaces, organization, component actions, and programming - were followed in order for the most part. The steps are not completely independent, however, so some iteration occurred. The program for MEMORY, for example, ended up differently than had been originally envisioned. This caused a change in the internal organization of the MEMORY group (step 3) and in the way in which it was accessed (step 5). Of importance though is the fact that the change to MEMORY did not affect any other aspect of the organization. The need for
and role of memory remained unchanged. The only external
effect of the reorganization was a change in the syntax
of statements which access the memory.

6.2 Evaluation of Modula

Overall, Modula proved to be an excellent tool for the
design of the message switching system. In at least four
respects, Modula made it easy to go from the specification
to the implementation of the system.

First, the building blocks of Modula - processes and
modules - were both appropriate and easy to use. The clarity
and reliability of an implementation is obviously affected
by the implementation language. In my opinion, Modula is
the best existing language for multiprogramming systems.
Interface modules provide exclusion of access to shared
variables and make it easy to pass data as records. They
make it easy to both describe process interfaces and reason
about the effect of process interaction. And device modules
provide a natural, encapsulated means for describing device
interfaces. The power of the language very definitely in-
creased my productivity and enabled me to program the entire
system in about 15 days. Without access to a compiler, I
have undoubtedly made numerous unintentional syntactic and
logical errors. Having described this design and implementa-
tion to numerous people however, I am convinced that no
major or global errors exist. To go from this report to a
working system should only require debugging each individual
component. Some changes may be required in order to tune
the system for good performance, but most of these changes
should occur only within device modules (e.g. changes to
buffer sizes). It is obviously a guess, but I think that
this system could be made operational in at most a very
few months, given a compiler.

The second value of Modula is the power of device
modules as a method for interfacing to IO devices. For
one, device modules provide users with a natural, pro-
cedural interface to devices. For example, the subscriber
control processes (SINPUT and SOUTPUT) input and output
characters by calling read and write procedures. Details of
buffer management, IO synchronization, and interrupts are
hidden from the user. It was also possible to schedule
future access to auxiliary memory (within AUXMEM) in parallel
with disk access. In fact, interface and device modules
made possible the clean separation of the file system functions
in MEMORY (e.g. management, memory mapping, and deadlock
prevention) from the device functions in AUXMEM (e.g. scheduling
and buffer management). A final advantage of device modules
is the ability to use different device access methods in
SUBSCRIBERS and TRUNKS. The SUBSCRIBER modules used the doIO
instruction because of the assumption that terminals give
interrupts. The TRUNK modules on the other hand used timing
signals to synchronize IO because of the assumption that
trunk lines periodically provide or expect characters. Pro-
gramming these differences is straightforward and they are
hidden from users of SUBSCRIBERS and TRUNKS.

A third feature of Modula, the ability to export read only variables from modules, made it easy to implement output pre-emption. The output control processes could test for pre-emption by merely checking a flag set by LINEOUTPUT. If this flag were not accessible outside of LINEOUTPUT, output controllers would have had to call a LINEOUTPUT procedure in order to interrogate the flag. In addition, note that changing the grain of pre-emption, namely the frequency at which the pre-empt flag is checked, merely involves moving the when statement in output controllers. Also note that LINEOUTPUT can easily restart pre-empted messages because they remain on a linequeue until completely processed.

The fourth and final positive aspect of Modula is its apparent efficiency. The amount of storage and execution time required by the Modula kernel (which implements processes, exclusion, signals, and IO completion) is minimal [10]. Most of the storage space and execution time used in the message switch should result from functions in the system itself.

In spite of its power, Modula is slightly deficient in three respects. First, because processes and modules cannot appear in type declarations, subscriber and trunk components must be declared for each terminal and trunk line. This leads to a tremendous expansion in the size of the program listing even though each subscriber group or each trunk group
is the same. Obviously, the same storage space is required in either case so efficiency is not affected. Readability and clarity is, however. Along the same line, the systems as defined requires a very large number of processes - four for each subscriber and trunk group. Since a typical system has approximately 50 local subscribers and 3 remote trunk lines, this results in over 200 processes. How this could affect efficiency is unknown.

A second, although minor, deficiency of Modula is the lack of queues or a way to construct them easily. As pointed out in the previous chapter, a queue can be implemented by a module. But a separate module is required for each different type of queue. It would be nice to have generic (polymorphic) procedures which operate on any type of queue. In parallel systems, queueing occurs frequently within interface modules so a general queueing facility, or the tools with which to construct one, would be quite useful.

The final deficiency of Modula as a language for designing fairly large systems is the use phrase. In Modula, use is optional; if omitted a module has access to all globally defined objects not renamed in the module. At a minimum, however, use should be required. A module should explicitly state what it is using so that a compiler can catch invalid accesses. Better yet, use should be replaced by a grant statement which specifies, when a process or module is declared, which other objects can access it. At present, Modula
puts access control in the hands of the user of an object; it should instead be put in the hands of the object's declarer in order to insure that the object is adequately controlled. This distinction has little effect in a small system but in a large system, especially one implemented by many people, it can have a great effect upon reliability. When one object is changed, it should be clear, and explicitly stated in the program, which other objects are affected by the change.

The above three deficiencies of Modula can and should be removed in a language for general system programming such as that proposed for the Department of Defense. They are relatively minor, however, and should not obscure the fact that Modula is the most powerful tool currently available for the design and construction of structured multi-programming systems. The message switching system in this report is but one example of Modula's utility.

Acknowledgement

Sera Amoroso and Derek Morris conceived of this project and served as sounding boards for the design as it was developed. Their assistance is greatly appreciated.
Bibliography


