SINTRAN III
Real Time Guide

ND-60.133.02
Revision A

SINTRAN III
Real Time Guide

ND-60.133.02
Revision A

NOTICE

The information in this document is subject to change without notice. Norsk Data A.S assumes no responsibility for any errors that may appear in this document. Norsk Data A.S assumes no responsibility for the use or reliability of its software on equipment that is not furnished or supported by Norsk Data A.S.

PRINTING RECORD

Printing	Notes
01/82	Original Printing
12/82	Version 02, Sintran H Version
02/83	Revision A:
	The following pages have been revised or added:
	v, vi, vii, ix, x, 43, 54, 54A, 60, 62, 63, 64, 65, 82, 257, 258

SINTRAN III Real Time Guide
Publ.No. ND-60.133.02 Rev. A
February 1984

Norsk Data A.S
Graphic Center
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Manuals can be updated in two ways, new versions and revisions. New versions consist of a complete new manual which replaces the old manual. New versions incorporate all revisions since the previous version. Revisions consist of one or more single pages to be merged into the manual by the user, each revised page being listed on the new printing record sent out with the revision. The old printing record should be replaced by the new one.

New versions and revisions are announced in the ND Bulletin and can be ordered as described below.

The reader's comments form at the back of this manual can be used both to report errors in the manual and to give an evaluation of the manual. Both detailed and general comments are welcome.

These forms, together with all types of inquiry and requests for documentation should be sent to the local ND office or (in Norway) to:

Norsk Data A.S
Graphic Center
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Preface

THE PRODUCT

This manual documents the real time facilities in the following products:

SINTRAN III VSE - version H	ND-10174
SINTRAN III VSX - version H	ND-10175
SINTRAN III VSE-500 - version H	ND-10049
SINTRAN III VSX-500 - version H	ND-10050

The manual is revised for version I of the above products. The product SINTRAN III RT is documented in the manual SINTRAN III RT User's Guide (ND-60.082) (not yet available).

THE READER

This manual is intended for application programmers who need a thorough description of the real time programming facilities in SINTRAN III. These include the software interface to and control of external devices, time dependent programs, synchronous interactive programs and programs which are executed at regular intervals.

Systems programmers who are going to study the SINTRAN III source code should be familiar with the contents of this manual.

PREREQUISITE KNOWLEDGE

The reader should be familiar with the contents of the manual:

SINTRAN III TIMESHARING/BATCH GUIDE (ND-60.132)

A reading knowledge of Fortran, and experience in the language to be used for real time programming, eg., Fortran, Planc, Pascal, NPL, or MAC, is assumed. Those monitor calls available in Fortran are illustrated in Fortran, but MAC examples are also included. MAC is the ND-100 assembly language. It is described in the manual MAC User's Guide (ND-60.096).

Familiarity with real time programming and concurrent processes is helpful, but not essential.

THE MANUAL

This manual describes commands and monitor calls used for real time programming under SINTRAN III. The additional facilities available in ND-500 computer systems are not covered by this manual.

The functions are ordered according to functional category rather than alphabetically as in the SINTRAN III REFERENCE MANUAL (ND-60.128). The most important commands in the SINTRAN III REAL TIME LOADER are described. Complete documentation of this subsystem is found in the manual SINTRAN III REAL TIME LOADER (ND-60.051).

Monitor calls and library routines also available in background programs are not necessarily illustrated by examples. The user should consult the SINTRAN III REFERENCE MANUAL (ND-60.128) for details of these calls.

RELATED MANUALS

The following manuals are related to the contents of this manual:

SINTRAN III TIMESHARING/BATCH GUIDE (ND-60.132)
SINTRAN III COMMUNICATION GUIDE (ND-60.134)
SINTRAN III REFERENCE MANUAL (ND-60.128),
SINTRAN III SYSTEM SUPERVISOR (ND-60.103),
SINTRAN III REAL TIME LOADER (ND-60.051)
NRL - ND RELOCATING LOADER (ND-60.066)
FORTRAN - ND FORTRAN Reference Manual (ND-60.145)
MAC User's Guide (ND-60.096)
SINTRAN III RT User's Guide (ND-60.082)

NOTATIONS USED IN THE MANUAL

In the examples, user input is underlined. Where SINTRAN III @CC commands are used in examples to comment other commands, these are not underlined, even though they are not returned as a response from the computer. Examples are given in UPPERCASE letters, however, lowercase letters are accepted in SINTRAN III commands and in the source program if the PLANC, PASCAL or FORTRAN-100 compilers are used. This does not apply to the old FTN compiler.

Octal numbers are given in the form 377B. SINTRAN III commands expecting decimal input also accept octal numbers if followed by a B. The term K, as in 128 Kwords, indicates 1024 (= 2 ** 10).

In command parameter descriptions, the parameters are enclosed in angular brackets, eg., <parameter>. Parameters which have default values are enclosed in parentheses, eg., (<parameter>). The default value is used if a null parameter is supplied (two successive commas or a prompt answered with carriage return).

Parameters are separated by spaces in command descriptions. When used, parameters should be separated by comma or space.

Alternatives in parameter descriptions are separated by slashes, /. Uppercase letters indicate that one or other value must be specified. For example, <WP/NP> indicates that "WP" or "NP" must be given.

The term "word" refers to 16 bit words, unless 32 bit words are specified. The term "byte" refers to 8 bit bytes. The upper (left) byte in a word is the most significant half of the word, the lower (right) byte is the least significant half.

The bits in a byte or word are numbered from the the right starting at 0 (the least significant bit). Bit 7 (byte) or 17B (word) is the leftmost (most significant).

The notation n:m is used to denote the numeric range from n to m inclusive.

CHANGES FROM PREVIOUS VERSION

The main changes from the previous version affect the use of the system included segments and the layout of the RT description. A new feature allows symbolic names to be defined for segments.

MAIN CONTENTS

Section Page

Section Page

Section Page

Section Page

Section Page

Section Page

Section Page

1 INTRODUCTION

Execution of real time programs is directly affected by external conditions. It may be synchronized with physical events and is in direct communication with the "outside world". The following are typical applications of real time programs:

* The temperature in a chemical process is continously monitored and if it falls below a specified limit, a heating element is activated to raise the temperature to an acceptable value.

* When a sensor reports that a car is approaching the road crossing, preparations should be made to switch to green light in the direction of travel of the car.

* If the core of a nuclear reactor is overheated, precautions must be taken immediately to prevent melt-down.

* Air humidity, temperature and wind speed at an observation post are read every ten minutes and stored for later analysis.

* When a telephone call is made through one exchange, a program in another exchange is activated to route the call to the receiver.

* The calculation of pi to a million digits should not slow down work for ordinary users, but when no others are making use of the CPU the calculation is allowed to progress.

* To save energy, the thermostat regulating office temperature is adjusted down at six every night and reset to normal at six every morning.

These are examples of problems that can be solved with programs which may:

* receive input from and control external devices
* have higher urgency than usual
* are started by external signals (interrupts)
* are periodically executed
* affect execution of other programs
* have a lower priority than other programs
* run at a specific time of day

Such programs are called real time (RT) or foreground programs, as opposed to timesharing background programs. They have requirements beyond those of ordinary background programs, and can perform operations not permitted for background programs. However, foreground programming requires much more from the programmer, as abuse of RT privileges may have disastrous effects on the computer system as a whole.

As an example of a real time application, a set of small Fortran programs is included below.

It illustrates simple use of foreground facilities, and there are numerous other ways to solve the same problem. The mechanisms used are explained in later chapters.

The set of programs does the following:

- The program REQUEST is started at time 0800 (start at specific time).
- It seizes terminal number 52 (reserve an external device).
- The message 'IMPORTANT MESSAGE - PLEASE ENTER ID' is printed on that terminal (controlling an external device).
- REQUEST sets up RESPONSE to be activated as soon as an interrupt is received from terminal 52 (affecting execution of other programs; reacting to interrupts).
- REQUEST also sets up PROMPT for execution every 10 seconds (repetitive execution).
- Every time PROMPT is executed, the message 'Hurry up!' is printed on terminal 52.
- REQUEST becomes dormant for 60 seconds. After that time it stops PROMPT and checks if a response has been received. This has been flagged by RESPONSE, setting the logical variable RCEEVD in RTCOMMON to .TRUE. (stopping another program; communication between programs).
- If no message has been received, WARN is started and the priority of this program set at 140 - higher priority than any ordinary terminal (high priority program).
- WARN prints the message 'No response from terminal 52' on the system console.
- After this, terminal 52 is returned to whichever program controlled it before REQUEST was started (releasing an external device).

```
C CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM REQUEST,40
      INTEGER WHDEV
      EXTERNAL WHDEV
      EXTERNAL RESPONS, PROMPT, WARN

      COMMON /RCV/RCEEVD
      LOGICAL RCEEVD

      INTEGER WHOHAS

      WHOHAS= WHDEV(52,1)
      CALL PRLS(52,1)
      CALL RESRV(52,1,0)
      WRITE(52,100)
 100  FORMAT(1X,'IMPORTANT MESSAGE - PLEASE ENTER ID',//)
      CALL RT(RESPONS)
      CALL INTV(PROMPT,10,2)
      CALL RT(PROMPT)
      CALL HOLD(60,2)
      CALL ABORT(PROMPT)
      IF (.NOT.RCEEVD) THEN
        CALL PRIOR(WARN,140)
        CALL RT(WARN)
      ENDIF
      CALL PRLS(52,1)
      CALL PRSRV(52,1,WHOHAS)
      END

C CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM RESPONS, 40

      INTEGER  WHDEV
      EXTERNAL WHDEV, PROMPT
      CHARACTER*1 CH

      COMMON /RCV/RCEEVD
      LOGICAL RCEEVD

      INTEGER WHOHAS

      CALL PRLS(52,0)
      CALL RESRV(52,0,0)
      READ (52,110) CH
 110  FORMAT(1A)
      RCEEVD= .TRUE.
      CALL ABORT(PROMPT)
      CALL RT(REQUEST)
      END
```

```
C CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM PROMPT, 50

      INTEGER WHDEV, WHOHAS
      EXTERNAL WHDEV

      COMMON /RCV/RCEEVD
      LOGICAL RCEEVD

      WHOHAS= WHDEV(52,1)
      CALL PRLS(52,1)
      CALL RESRV(52,1,0)
      WRITE(52,120)
 120  FORMAT(' HURRY UP!',/)
      CALL RELES(52,1)
      CALL PRSRV(52,1,WHOHAS)
      END

C CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM WARN, 50

      CALL PRLS(1,1)
      CALL RESRV(1,1,0)
      WRITE(1,130)
 130  FORMAT(1X,'No response from terminal 52',//)
      END

C CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Compilation and loading procedure:

```
@FORTRAN
ND-100 ANSI 77 FORTRAN COMPILER - NOVEMBER 24, 1981

FTN:COMPILE EXAMPLE,EXAMPLE:LIST,EXAMPLE

- CPU TIME USED: 3.1 SECONDS.  82 LINES COMPILED.
- NO MESSAGES
- CODE SIZE=520 DATA SIZE=0 COMMON SIZE=1 STACK SIZE=32
FTN:EXIT

@RT-LOADER

REAL-TIME LOADER, SINTRAN III  -  VERSION G

*NEW-SEGMENT 270,,,,
*SET-RTCOMMON RCV
*LOAD EXAMPLE,,,
*LOAD FORTRAN-1B-LIB
*END-LOAD
*EXIT

@
```

Schedule REQUEST for execution at 8 a.m.:

```
@ABSET REQUEST,,,,8
```

At terminal 52 at 0800:

```
 IMPORTANT MESSAGE - PLEASE ENTER ID

HURRY UP!

HURRY UP!

RONNIE
```

The HURRY UP!s are printed at 10 second intervals. If no input were entered, after 1 minute would appear on the system console:

```
 No response from terminal 52
```

2 SYSTEM OUTLINE

This chapter briefly describes the most important points in Sintran RT programming. It provides an introduction for programmers who do not need to go into the details of e.g. interrupt handling and memory management. The reader can go from here to chapter 7, which explains how to load a program for real time execution and then go back to the detailed chapters later.

2.1 The user RT

The user RT can use the facilities for real time programming. Ordinary users cannot start real time programs. The user RT is the owner of files created and used by RT programs, unless another user name is specified when opening the file.

2.2 The user SYSTEM

The user SYSTEM is the most highly privileged user in the Sintran III system. SYSTEM can use all RT facilities and in addition can set system parameters using the @SINTRAN-SERVICE-PROGRAM and act as supervisor. Facilities only available to user SYSTEM are only described in this manual if they relate to real time programming; they are indicated as unavailable to user RT.

2.3 RT monitor calls

RT programs request services from the operating system through monitor calls, which are machine instructions temporarily transferring control to the operating system.

Most monitor calls are available as subroutine calls from Fortran and Planc. In MAC, the MON instruction is used. Various monitor calls are described in this manual.

The functions and services provided by monitor calls are usually available as commands that can be executed by a terminal operator. The term "call" is sometimes used to indicate either the monitor call or the command (e.g. "the ABORT call ...").

2.4 SINTRAN III commands

Logged in as user RT (or SYSTEM), an operator can control resources and RT programs in a manner quite similar to use of monitor calls in a program. In addition, there are a few operations which can only be performed by commands. Like monitor calls, commands cover a wide range of RT applications and are described at various points in this manual.

2.5 RT programs

An RT program is a set of instructions to be executed sequentially, requiring no parallel execution. Several RT programs may be active at once, (virtually) executing concurrently.

2.6 The RT description

Each RT program that has been loaded into a SINTRAN III system with the RT loader is described in an RT description. This contains all information required by the operating system to provide CPU time, memory space and access to peripheral devices for the RT program. The number of RT descriptions available in the system is limited and is determined at system generation time. The layout of the RT description is described in chapter 6.

2.7 Peripheral equipment interface

Peripheral devices are accessed from user RT programs through monitor calls. The device is identified by its logical device number and the operating system will perform the device dependent translation. Peripheral devices are described in chapter 10.

2.8 Datafields

The datafield contains that information about a physical device which is needed by the operating system. When a device is reserved for an RT program, the datafield is linked to the reservation queue of the program through link locations. To each datafield is also linked the RT descriptions of programs that have requested the device, but not yet acquired it.

Datafields are also used in the same way for internal devices. They are described in chapter 6.

2.9 Segment files

Although programs must be located in primary memory at execution time, they are stored on disk when not active, or when the number and size of active programs means they cannot all be in memory. RT programs and their data are stored on files named SEGFIL0:DATA, SEGFIL1:DATA etc. (Not reserved file names!). A system may have from one to four segment files, each containg a number of RT programs and data segments.

2.10 Segments

A segment file is divided into segments, which are images of contiguous parts of memory. They may contain one or more RT programs or may contain data. When a segment is needed by an RT program, or an RT program on the segment is started, the segment (or the required part) is copied to physical memory.

A segment has an integral number of 1024 word (2048 byte) pages, but in most cases it is treated as an indivisible unit by the operating system. It is located in a contiguous area in a segment file on disk, it is copied from disk directly into physical memory. Memory allocation during execution and protection applies to the segment as a whole. A segment may be between 1k (1024) and 64k (65536) words, in units of 1024 words.

2.11 The segment file bit map

Deletion segments no longer in use may leave open space in the segment file which can be used when creating new segments. A bit map is kept to indicate which file pages are used by segments and which ones are free for use. When a new segment is created, there must be a contiguous area of free pages at least as large as the segment.

2.12 The real time loader

The segment files, segments, RT programs and data areas in these files are maintained through the RT loader. All program code and data to be entered on a segment must be loaded through the RT loader. The memory allocation parameters, privileges and protection of a segment are set through RT loader commands.

The RT loader is an integral part of the SINTRAN III system. It is described in detail in chapter 7.

2.13 RTFIL:DATA

The file RTFIL:DATA contains all the information about symbols defined on segments in the system and global names such as RT program names etc. This information is used to arbitrarily link symbolic references with definitions during RT loading.

2.14 The linking table

During loading the RT loader maintains a temporary table of all symbols defined and all symbolic references encountered in the loaded modules. The symbols in this table are written to the RTFIL at the end of the load.

2.15 Memory management

Physical memory is not generally large enough to hold all active programs and their data. A mechanism is required to continiously check whether parts of a program are active or not active - whether their presence in physical memory are needed or not.

In many cases two programs require access to a common area without sharing the entire address space. A mechanism is required for splitting the address area into parts that can be handled separately.

A program may need special protection or privileges affecting the legality of each machine instruction. At execution time, a protection mechanism is effective, monitoring each executing instruction to trap security violations.

These problems are solved by the operating system through the memory management hardware described in detail in chapter 3.

2.16 Pages

Memory is divided into units of a page, blocks of 1024 (1k) 16 bit words. Physical memory and segment area are allocated in units of one page. A page is always contiguous in physical memory and on the disk.

2.17 Page tables

Although logically contiguous, the addressing area of a segment consists of pages scattered in physical memory. This permits selected pages to be retained in physical memory while others are written back to disk. This flexibility means the physical page address must be looked up in a page table for each instruction executed. This is performed by special purpose hardware in memory management system, which minimizes the time taken.

2.18 Memory map table

The contents of the page tables are specific for each segment. When a program in another segment starts execution, the relevant page tables must be loaded from the memory map table. The RT programmer does not see this table, it is maintained exclusively by the operating system.

2.19 Segment table

The segment table keeps information about memory allocation of a segment so the operating system can find pertinent data when placing the segment in physical memory. The programmmer does not see this table.

2.20 Ring protection

Each segment belongs to one of four rings, or classes, termed ring 0 to ring 3. The ring determines the set of instructions that may be executed by programs on the segment and the locations that may be accessed outside the segment. Ring 3 is the most highly privileged ring and is not available to user RT programs. Ring 2 is used by the operating system and RT programs using privileged instructions. Ring 1 is used by RT programs accessing RTCOMMON. Ring 0 is used by ordinary background programs and RT programs which do not use RTCOMMON.

Although each memory page has its own ring setting in the page tables, the RT loader considers the ring is a property of the segment and all programs on the segment reside in the same ring. Ring protection is described in chapter 3.

2.21 Page protection

The page tables contain three bits for each page: RPM, WPM and FPM, indicating Read, Write and (instruction) Fetch Permission. By default, all three bits are set, allowing all types of access. If the RPM bit is reset, data cannot be read from the page, if the WPM bit is reset locations in the page cannot be modified and if the FPM bit is reset words in the page cannot be executed as instructions.

The RT loader gives the page protection bits the same setting for each page in a segment, but different segments may have different protection settings. The page protection bits apply to all accesses, including those executed by instructions in the same segment and are independent of the ring protection. For a page to be accessible to a program, both ring and page protection bits must allow sufficient access.

Page protection is described in chapter 3.

2.22 The RT monitor

Resources in Sintran III are maintained by a set of operating system routines called the RT monitor. Requests for access to resources are routed to the RT monitor through a monitor call instruction. The RT monitor also provides system information and services through the same mechanism.

The RT monitor runs independently of user RT programs, executing at a higher interrupt level. This is described in chapter 3.

2.23 Queues

If several programs require access to a resource (e.g. external device, memory buffer, non-reentrant routine) that can be used by one at a time, the second and following programs must wait in a queue until the resource is released by the reserving program. There are different queues for each resource. A program in a queue is idle and does not require any CPU time. It cannot enter a second queue. (The time queue is treated as a special case and is not affected by this limitation.)

A queue is implemented as a singly linked list, linked through "link locations" in the RT description, datafield, etc.

All queues are maintained by the RT monitor. They are described in chapter 6.

2.24 Execution queue

The execution queue holds programs waiting only for CPU time or execution of an I/O transfer. Programs are entered and removed from the queue by the operating system. An executing program may be forced to give up the CPU, if a program of higher priority enters the queue. The program executing, is also in the execution queue.

2.25 Waiting queues

Each resource may have a queue containing all programs which have requested the resource but not been given access to.

Unlike the execution queue, a high priority program cannot force a lower priority program to give up the resource. But the queue is ordered with respect to priority and the highest priority program in the queue will be granted the resource when it becomes available.

2.26 Reservation queues

With each active RT program there is a queue linking all resources reserved by this program. This queue will link datafields of the devices (internal or external) together.

2.27 Monitor queue

If the RT monitor handles two or more devices concurrently, the datafields of these devices are entered in the monitor queue.

2.28 Time queue

The time queue contains programs scheduled for future execution by an operator or a program and programs which execute periodically.

A program may be in the time queue and another queue at the same time. Presence in the time queue does not affect the execution of the program until the waiting period has expired, at which time the program is scheduled for execution.

2.29 Background RT programs

A terminal used for ordinary timesharing operations is controlled by a system included RT program with the name BAKnn, where 'nn' is a two digit number. The numbers are not necessarily in sequence. Background RT programs are given special treatment in allocation of CPU time, but are in most respects analogous to a user written RT program. See the time-slice mechanism explained later.

Batch processors are similar system included programs, with name BCHnn. 'nn' is 01 for the first batch processor, 02 for the second and so on.

2.30 Timeslicing

If not interrupted, an RT program that has started execution will execute until it completes, is interrupted by a higher priority program or gives up the CPU voluntarily. The latter may be due to a request for a resource that is not immediately available.

If a program with a higher priority enters the execution queue, the lower priority program is forced to give up the CPU. Several programs will interleave their execution in the CPU if they have different priorities.

The priority of background programs is modified dynamically, effectively interleaving execution of several programs. RT programs, however, will have a static priority unless explicitly modified.

2.31 Demand/Nondemand segments

The swapping system does not necessarily bring the entire segment from disk into memory as soon as the segment is accessed - only those pages actually used are fetched. This is called demand paging.

Elapsed time for a program using a demand segment varies, depending on whether the required pages are in memory or not. To obtain a more predictable response time, segments used by RT programs are by default nondemand segments. As soon as a nondemand segment is taken in use, all pages of the segment are brought into memory. RT programs may also use demand segments, but demand paging must be requested explicitly.

2.32 Fixing a segment in memory

An ordinary segment is not copied from the disk to physical memory until one of the RT programs using it is prepared for execution, and (in case of a demand segment) only those pages of the segment actually in use will be in memory. When a page is needed which is not in memory, it is fetched from disk.

The time required to access the disk may be too long for time critical tasks and response times cannot accurately be predicted. To achieve a short and well defined response time the segment may be declared as fixed, which means that all its pages are held in memory until explicitly removed (unfixed). The segFixing a segment is rather costly in terms of system load and should be used with care. Fixed memory allocation is described in chapter 8.

2.33 Segment sharing

Two or more RT programs may share a segment in order to exchange data because they execute the same routines. A program may have its private routines and data in one segment and share routines and data with other programs in another segment. Segment sharing is described in chapter 14.

A mechanism is also available to allow programs to share a segment provided they do not make modifications to it. As soon as a program makes a modification, it receives its own private copy of the affected page. Other programs continue to read from the original, unmodified page. Such segments are termed reentrant segments. The mechanism is described in chapter 15.

2.34 Program communication

Different RT programs may want to exchange messages, to synchronize in time or to agree on the use of a common resource. The operating system provides several mechanisms for exchanging timing signals, small and large amounts of data.

2.34.1 Semaphores

Semaphores are simple timing signals, used to synchronize programs or to flag e.g. a non-reentrant routine or a common data area as reserved or free. Semaphores are described in chapter 11.

2.34.2 Internal devices

Data exchange channels may be accessed in a manner similar to files, using the same monitor calls. The data transfer is of course much faster. Normally, one RT program will write to the internal device, while another RT program reads from the same device. If nothing is written to the device, the reading program enters a waiting state until there is something to read. If the internal device's buffer is full, the writing program may enter a waiting state until a program reads from the buffer. Chapter 12 describes internal devices.

2.34.3 RTCOMMON

Cooperating RT programs on different segments may need extremely fast communication, e.g. for resource protection. An area of physical memory may be set aside when Sintran is loaded for data that are permanently kept in memory and never swapped out. This area is called RTCOMMON and is accessible to all RT programs. Communication is immediate; when one program stores a new value to a location in RTCOMMON, the next load from the same location will fetch the new value, regardless of which program performs the load.

In most systems, the size of RTCOMMON is small and RTCOMMON variables serve as flags to protect larger areas in ordinary segments or peripheral devices. RTCOMMON is described in chapter 12.

2.34.4 Files

All files opened by an RT program may be accessed by other RT programs. The default user name when opening a file is RT. Unlike background programs, RT programs may start a file transfer in parallel with program execution. Chapter 13 describes the use of files from RT programs.

2.34.5 Communication with background programs

Internal devices may be reserved and accessed for read and write by background programs while the other end of the channel is used by an RT program. If machine independence is required, the internal device may be named in the file system and used as an ordinary, sequential file. Also, any file may be accessed both by RT and background programs.

2.34.6 X-message XMSG

X-message is an optional part of SINTRAN III for general communication between background and RT programs, drivers or direct tasks, or any combination of these. XMSG is described in the SINTRAN III Communication Guide (previously called Special I/O guide) and is not described in detail in this manual. A short introduction is given in chapter 20.

2.35 Reentrant programs

For two programs to share routines without influencing each other, these routines must be reentrant. A reentrant routine may be reactivated (possibly by another RT program) before it has terminated. The basic requirement for achieving reentrancy is separation of code and data, usually by a stack mechanism. High level languages used for RT programming generate reentrant code (for Fortran this is optional). Reentrancy is described in chapter 15.

2.36 Recursive routines

A recursive routine calls itself, either directly or indirectly. Recursion is commonly used to solve certain mathematical problems and to handle advanced data structures (such as traversing a tree). The routines must be reentrant. Recursion is described in chapter 15, Reentrant systems.

2.37 Interrupts

The "outside world" signals to a program that its attention is required by sending an interrupt signal. A connection between the interrupting device and the program may be set up through a monitor call. Chapter 4 describes the interrupt system.

For I/O operations on standard devices, the RT program need not handle the interrupt itself. It requests input or output by executing a monitor call and the device driver, which is an integral part of the operating system, takes care of the interrupt handling and transfers data to or from the user program's memory or registers.

2.38 Clock interrupts

Timing signals are treated in the same manner as an interrupt from an external device, but are handled by a system program that checks whether any other program is waiting for the recently arrived clock signal. This is used by periodic programs or those scheduled for execution at a specific time.

2.39 Direct tasks

In certain applications it is desirable to run programs independently of the SINTRAN III operating system. Such programs are termed 'direct tasks'. These may run at a higher hardware priority level than other programs (including or excluding the operating system routines). No operating system support is provided and the ordinary protection mechanisms are not in effect.

Direct tasks are described in chapter 17.

2.40 Languages for real time programming

In principle any language can be used for RT programming, but not all are equally suited. All facilities are available in MAC and the programmer communicates with the operating system directly through monitor calls. Assembler language is rarely used other than for programming drivers and other extremely time critical tasks, due to the poor data abstraction and structuring facilities.

NORD-PL is a low level language that serves as convenient shorthand notation for MAC code. A limited number of data and flow structuring tools are available. NORD-PL is translated to MAC source code before it is assembled by the ordinary MAC assembler. A knowledge of NORD-PL is required to understand the operation of SINTRAN III routines and drivers, but most programmers need not concerned themselves with it.

Fortran is the 'classical' higher level language for RT programming on ND computers. The ANSI 77 standard Fortran (FORTRAN-100) is extended with the capability to generate reentrant code and may also be used for recursive routines. Most monitor calls are available as Fortran subroutines or functions.

Planc is the newest system programming language for ND computers. It provides high level data structuring facilities, flow control constructs, modularization tools and exception handling. For time critical tasks, MAC code may be coded inline in selected parts of the program. The Planc language is the same for the ND-100 and ND-500 computers.

Pascal is a well known and widely used language that may be used for RT programming. Data structuring and flow control mechanisms are excellent. Time critical tasks and special monitor calls may be coded in MAC and linked as external routines. The most commonly used monitor calls are available as external routines in the standard Pascal library. RT monitor calls are linked from the Fortran library.

ND **Basic** is a simple beginner's language. A few extensions to standard Basic features allow Basic to be used for RT programming. The majority of special RT functions are fetched from the Fortran library and linked in as external routines.

Cobol is not well suited for RT programming. Its I/O system may not be used in real time, all I/O and special RT routines must be fetched from the Fortran library and activated through the CALL statement.

A special purpose language for process control, **NODAL**, has been developed for ND computers at CERN. It is particularly suited in applications where operators interact with the computer, inspecting and modifying process parameters.

3 THE MEMORY MANAGEMENT SYSTEM

The memory management system consists of hardware dedicated to translation of program (virtual) addresses to addresses in primary memory (physical addresses). It also checks that the executing program has the required access rights to the addressed memory location.

(A virtual address is sometimes called a logical address. In this manual virtual is used throughout.)

The memory management system extends the capabilities of the ND-100 computer to include

1) an entire 64K virtual address space available to all users and each program independent of the physical memory size

2) two addressing areas - each of up to 64K words - per program, extending the virtual addressing area to 128K words

3) keeping only the active parts of a segment in primary memory by detecting access to swapped out pages

4) protection of parts of the addressable area from modification, from reading or from being executed as instructions

5) making possible an extension of the physical addressing range beyond 64K, thus allowing many users to have their programs in memory concurrently if the physical memory is large

6) permission for highly privileged programs to access pages belonging to those of lower privilege while access in the other direction is prohibited

3.1 The 64K virtual memory space

If physical memory addresses were equal to the address used in the program, two programs using the same addresses would read and write each other's data. The only way to prevent this would be to copy the entire memory space to a scratch area before the second user entered his program. In such a system, the time spent copying programs from disk to memory and back would probably take several times longer than actual program execution.

Instead a translation from program (virtual) address to memory (physical) addresses is performed by means of a translation table. When the memory management system is active all memory references are made through this table and there need not be any correspondence between the virtual and the physical address. This table is called a page table, for reasons explained below.

When one program has to give up its right to the CPU to make room for another, the program and data are retained in memory, but the entries in the page table are replaced. The new contents cause a virtual address in the second program to be translated to a physical address that does not conflict with the first program.

3.2 The page tables

If each memory location had its entry in the page table, the table would be at least as large as the program and nothing would be gained. Instead, virtual and physical memory are divided into pages - sections of the memory space of size 1K, 1024 16 bit words. Each page has one entry in the table, giving 64 table entries per virtual memory area. This splitting into pages is consistent throughout primary memory and the disk system and page boundaries coincide.

Pages that are logically adjacent in the virtual address space are not necessarily adjacent in physical address space, but may be scattered throughout memory in any order. However, each page is contiguous in memory. All virtual addresses belonging to the same page are found in the same physical page. The translation translates a virtual page number to a physical page number. The lower 10 bits of the address is the displacement from the page boundary (Displacement In Page) and is used directly with no translation.

Physically, the page tables are built from high-speed registers, causing an insignificant delay in the access to primary memory.

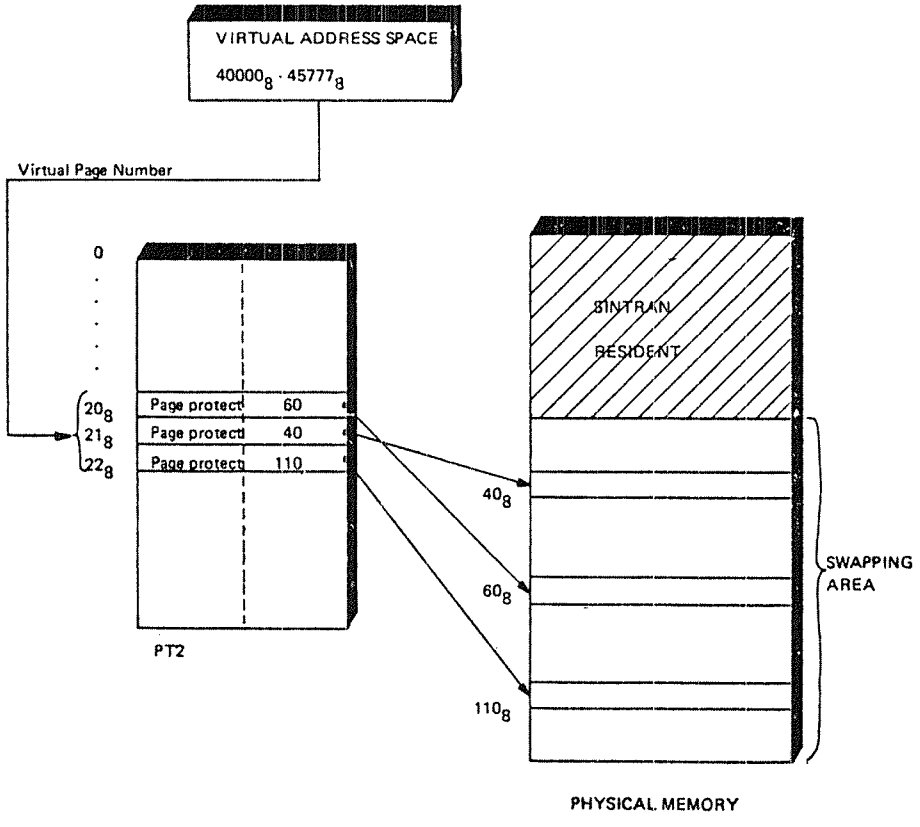


Fig. 1. Address translation

3.3 The use of the four page tables

The memory management system contains four page tables, PT0, PT1, PT2 and PT3, each capable of transforming any virtual page number in the range 0 to 63 to a physical page number anywhere in memory. Thus, four different addressing areas can be accessed without clearing the page tables.

PT0 is used by the operating system and should not be used by RT programs. The lower part of this table contains the "resident" part of SINTRAN III. Pages in this area are always in memory and are never written back to disk. They contain time critical operating system routines. Even the contents of the page table will never be replaced, thus these locations are not available to any other segment.

The upper part of the page table addresses the remaining operating system routines. The non-resident part of SINTRAN III is split into several segments. Each of these segments may be addressed through the upper half of PT0. When routines on a segment other than the current one are required, the contents of the upper half of PT0 must be replaced with the entries of the new segment.

Most RT programs use **PT1**. The major part of this table is available to any RT program, but includes the RTCOMMON area in the uppermost locations. RTCOMMON is a resident part of physical memory reserved for communication between RT programs. The initial size of this area is a parameter in system generation, but may be increased by the system supervisor. All programs using PT1 access the same physical locations when addressing within the RTCOMMON area.

PT2 is used by ordinary timesharing (background) programs. The entire 64K addressing area is available.

PT3 is free for use by any program. Background programs using the "2-bank" facility use PT3 for the alternative area. It may also be used by RT programs or direct tasks.

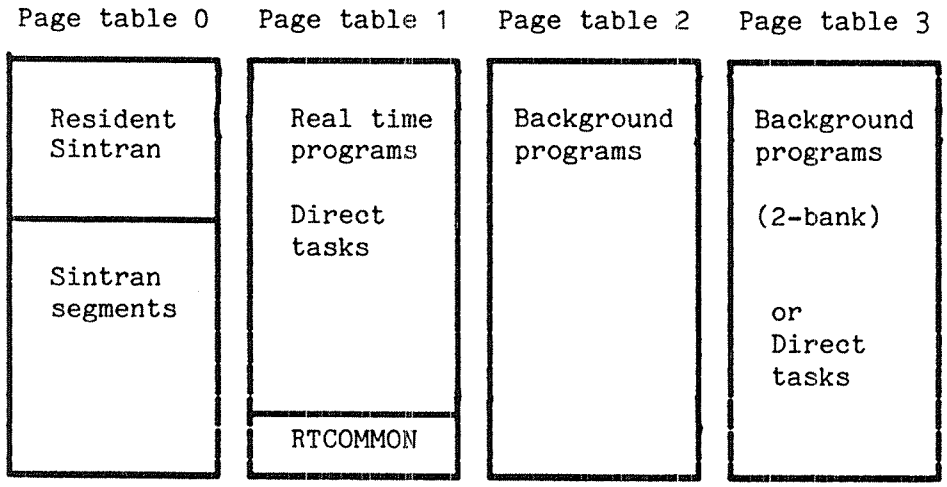


Fig. 2. Use of the page tables

Which page table to use depends on the segment. If several programs are located on the same segment, they will all use the same page table. One RT program may also use two segments concurrently, as described in the next section and these two segments may use different page tables.

Which page table to use is determined when the segment is loaded. The default table for RT programs is PT1, but the user may want to override this default for a number of reasons:

- In special cases, the RT loader may be used to load systems that is run as background processes. In that case, the segment should use PT2.

- If the "alternative page table" mechanism is used, the two segments to be used must use different page tables.

- If an RT program requires a full 64K address space or uses addresses that would (unintentionally) overlap RTCOMMON, PT1 may not be used and the segment must be placed on another page table.

- When a segment is used in conjunction with other segments, overlap is illegal. If the two segments are located on different page tables, no conflicts arise.

3.4 The page table entry

Each of the four page tables contains 64 entries, one for each virtual page in the address space. The virtual page number - the upper 6 bits of the address used in the program - selects one of the 64 entries. The entry describes the properties of one page and the translation from a virtual page number to a physical page number.

Each entry in the page table has the following format:

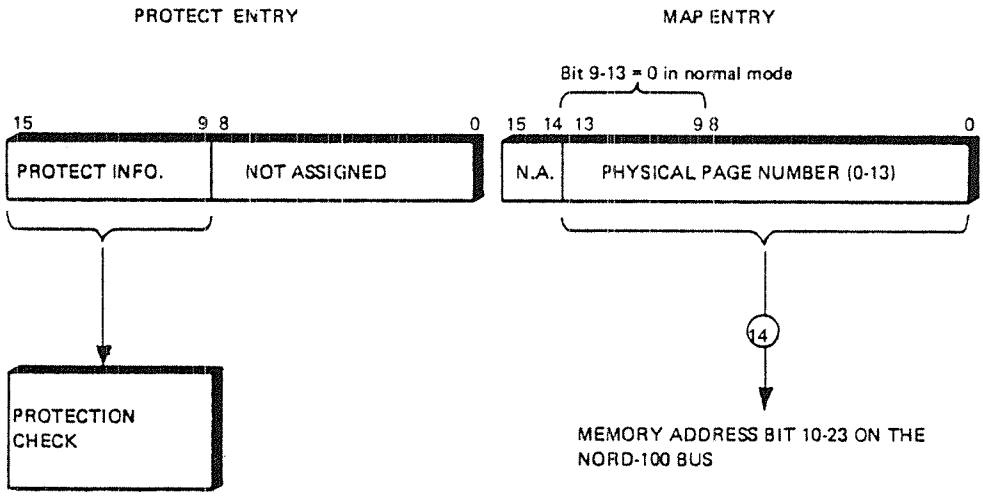


Fig. 3. ND-100 page table entry

The upper seven bits are used to trap unauthorized access to the page and special conditions requiring the attention of the operating system. A violation of access rights is immediately detected by hardware and causes an interrupt transferring control to the operating system. If an interrupt occurs, the instruction causing it will not be executed.

3.4.1 Page protect bits

Bits 15, 14 and 13 determine permitted access to the page. The WPM bit, if set, allows store operations to the page; if reset, write is not permitted. The RPM bit, if set, allows read (load) operations; fetch is not affected by this bit. The FPM bit, if set, allows words in the page to be executed as instructions; if reset, only data accesses are allowed.

Any combination of WPM, RPM and FPM is legal. If all three bits are reset no access is legal.

The page protect bits are discussed below in the section on protection mechanisms.

3.4.2 The Written In Page (WIP) bit

Bit 12, the WIP bit, is reset by the operating system when the page is read from the disk and is automatically set by hardware when the first write (store) operation is performed. It is used by the segment management system (see below) to determine whether a page has been modified and must be written back to disk, or can be overwritten by another page when required without any write back.

3.4.3 The Page Used (PGU) bit

Any access to the page (read, write, fetch) causes hardware to set bit 11, the PGU bit. Like the WIP bit, this bit is used by the segment management system to select a candidate as "victim".

3.4.4 Ring bits

Bits 10 and 9 determine which ring the page belongs to and are part of the protection system. Instructions in a given ring may access pages in an equal or lower ring, but not in higher rings. The rings are termed ring 0, ring 1, ring 2 and ring 3. Ring 3 is the highest (most privileged). The ring also determines the instruction set available to programs in the ring.

The ring protection system is discussed in the section below on protection mechanisms.

3.4.5 The Physical Page Number (PPN)

The physical page number is used in the actual translation process and gives the address in primary memory of the virtual page number selecting the entry. In the ND-100 the Physical Page number is a 14-bit value, allowing up to 16 384 physical pages, 32 megabytes.

The ND-100 may be operated in a NORD-10 compatible mode, using only a 9 bit physical page number. Bits 9 to 14 of the page number are then zero. One megabyte (512K words) of physical memory may be addressed. This mode is called the normal addressing mode and is default when the CPU is restarted. The 32Mb mode is called extended addressing mode. The extended addressing mode is entered by executing a SEX machine instruction (Set EXtended address mode), normal addressing is entered by a REX machine instruction (Reset EXtended address mode). (These instructions should not be used by any user RT program and are not permitted in ring 0 or 1.)

The NORD-10 computer uses a 9 bit physical page number, located in the lower nine bits in a 16 bit page table entry. The SEX and REX instructions are not available.

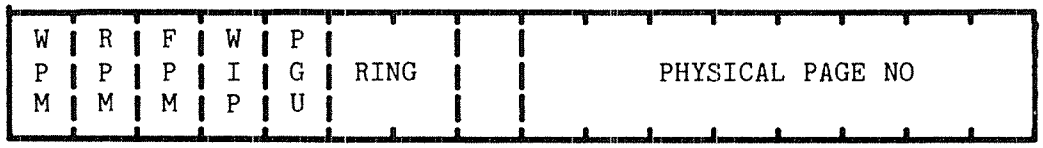


Fig. 4. NORD-10 page table entry

3.5 DMA devices and the memory management system

Disks, communication devices etc. usually operate through Direct Memory Access - DMA. The device controllers access physical memory directly and no translation is performed by the memory management system. Read or write permission is not checked and ring protection does not apply. The WIP and PGU bits are not affected by operations performed by DMA devices.

RT programs usually request I/O from such devices through the operating system, which initiates a data transfer to or from a page that is invisible to the RT program. When the transfer is complete, the data is copied to the program area using the memory management system under full control of the protection mechanisms. (This is in system documentation termed the "window mechanism".)

Programs dealing directly with DMA devices require special considerations:

The segment involved in the data transfer must be fixed in memory. If not, there is no guarantee that the affected page is in memory when the DMA device performs a data transfer.

In general, the physical memory address of the segment must be known and the segment must be located in a contiguous part of memory. This means that the FIXC call must be used to fix the segment (see section 8.2.5).

If the DMA device writes to the segment, the WIP bit must be explicitly set, to force the affected pages to be written back to disk when the segment is unfixed. This is done by specifying WP as the last parameter of the *NEW-SEGMENT command (see section 7.4.1).

3.6 Protection mechanisms

If all programs were error free and programmers never misused system resources, protection mechanisms would probably be unnecessary. However, errors in RT programs are inevitable during program development and may be catastrophic to the system as a whole.

Two protection systems are provided by the memory management system, the page protect system and the ring protect system. They are independent and either one may make a memory location unaccessible.

The page protect system limits the kind of access allowed to the page - reading data, writing data or executing as instructions. The ring protect system places each page in one of four rings, where access is permitted to pages in lower rings, but not to higher rings. Also, a set of instructions not normally available may be executed in ring 2 and 3.

3.6.1 The page protection system

Page protection consists of the WPM, RPM and FPM bits in the page table entry. Although a page has its own entry in the page table, all pages in a segment have the same page protection.

The page protect bits are set as a parameter to the *NEW-SEGMENT command in the RT loader. R, W and F indicates Read, Write and Fetch permit respectively and any combination is legal. E.g. to allocate segment number 210 with fetch permit only (using defaults for the remaining parameters), the command would be

```
*NEW-SEGMENT 210,,,
PROTECTION BITS: F,,
```

If the default value of the "protection bits" parameter is used, all types of access are permitted (WRF). The protection applies to the entire segment.

Non-default values of the protection bits are primarily used in systems with two or more segments. The segment containing the instructions should then be given Fetch permission only (as in the example above). A segment containing data that should not be modified (e.g. constant tables) should be given Read permission only. No purely data segment should have the Fetch permit bit set.

Observe that resetting the Read permit bit prohibits P-relative load instructions. It has been common practice in MAC assembler programming to address literals (constants) relative to the current program counter, by use of the MAC command)FILL. Although logically constants, hardware treats these locations like variables, prohibiting load operations.

Indirect access where the address is found in a Fetch access only page is allowed provided the data location to which it points is in a page permitting data access. In other words, the)FILL command may be used provided literals allocated by)FILL are used for indirect access only, as in LDA I (LIT .

Most high level language compilers place literals in the data bank and therefore create no problems even if the program segment is given fetch permit only.

3.6.2 The ring protection system

A page belongs in one of four rings. The ring determines which other pages may be accessed; only pages belonging to the same or a lower ring may be accessed. The normal usage of the rings is

Ring 0 - background programs, RT programs,

Ring 1 - RT programs using RTCOMMON, but no privileged instructions, database systems
Ring 2 - The operating system, including the File and I/O system, ND-Net etc.
RT programs executing privileged instructions
Ring 3 - Not used (intended for operating system kernel)

All pages in a segment belong to the same ring, determined by the second parameter of the *NEW-SEGMENT command in the RT loader. To start loading segment 211 and declare this segment to run in ring 2, the command would be (using defaults for the remaining parameters):

```
*NEW-SEGMENT 211
RING: 2
SEGMENT TYPE: ,,,,
```

The default ring is 0. Ring 0 segments using page table 1 may not use the addresses covered by the RTCOMMON area, otherwise the program is aborted with the error message RING VIOLATION.

Segments in ring 1 using page table 1 will access RTCOMMON in the uppermost addresses (see chapter 12). Ring 1 must be specified explicitly if the RTCOMMON area is used. The instruction set is equivalent to that permitted for background programs. Locations in ring 0 and 1 may be accessed.

Segments in ring 2 may access any location in ring 0, 1 or 2 (including RTCOMMON in ring 1). The instruction set is expanded with a set of privileged instructions:

```
  IOF   - Turn interrupt system off
  ION   - Turn interrupt system on
  POF   - Turn paging off
  PON   - Turn paging on
* PIOF  - Turn interrupt and paging off
* PION  - Turn interrupt and paging on
* LWCS  - Load writable control store
  WAIT  - Give up priority, reset current PID bit
  IDENT - Identify interrupt
  IOX   - Input/output
* IOXT  - Input/output
  TRA   - Transfer internal register to A
  TRR   - Transfer A to internal register
  MCL   - Masked clear of register
  MST   - Masked set of register
  LRB   - Load register block
  SRB   - Store register block
  IRW   - Inter-register write
  IRR   - Inter-register read
* REX   - Reset extended address mode
* SEX   - Set extended address mode
* EXAM  - Memory examine
* DEPO  - Memory deposit
* OPCOM - Start MOPC
* LDATX - Load A from physical memory
* LDXTX - Load X from physical memory
* LDDTX - Load AD from physical memory
* LDBTX - Load B from physical memory
* STATX - Store A in physical memory
* STZTX - Store zero in physical memory
* STDTX - Store AD in physical memory
```

If any of these instructions are executed by ring 0 or ring 1 programs, an internal interrupt occurs and the operating system terminates the program. The instructions marked with an asterisk ("*") apply to the ND-100 only and are not available in the NORD-10. A program may test bit 14B in the status register to determine whether it is executing on an ND-100 or a NORD-10. The bit is reset on a NORD-10 CPU, set on an ND-100 CPU.

Ring 3 is intended for the operating system kernel. (Currently, SINTRAN III does not use ring 3). The full instruction set and all of memory are available to segments in ring 3; these may not be loaded by the RT loader. (On the NORD-10 front panel the ring 3 indicator will flash when paging is turned off, and does not accurately reflect the current ring.)

Users of a device may be limited to programs of a certain minimum ring. This is indicated by the two lowest bits in the TYPRING subfield in the datafield of the device (see section 6.3). These two bits are checked against the ring of the requesting program when the device is reserved. If the ring of the program is lower, the reservation is denied.

An RT program belongs to a ring that is independent of the segments it uses, but it is not allowed to access a segment in a higher ring than itself. The program is given the ring protection of the default load segment when the program is loaded, and cannot later be modified (except by patching the RT description table in resident Sintran).

The current ring is determined by the Paging Control Register (see section 3.7.1) on the current level. The ring bits in this register are equal to the ring of the executing program. When the program starts execution the ring is fetched from the ACTPRI word in the RT description.

3.7 The "Alternative Page Table" mechanism

Most programming languages, particularily block structured "Algol-like" languages, maintain a strict separation of code and data. This helps prevent the program from being accidentally modified if the program is in error and it prevents data from being executed as instructions. Also, the same set of instructions can be applied to several sets of data.

To take full advantage of the separation of code and data, code may be put on one segment and data on another. Thus, several variants of the data segment may be used with the program segment.

If the two segments are placed on two different page tables, there is no way the segment boundaries can be crossed accidentally. It is physically impossible to execute data or to read/write instructions. Hardware may be set in a mode to distinguish between data accesses and fetching instructions, addressing through different page tables.

All P-relative addressing is assumed to be program addresses, B and X relative addresses and indirect references are data. In most cases program addresses are either jump labels or read-only constant values. (Most compilers will address even constants in the same manner as variables loaded in the data segment).

3.7.1 The Paging Control Registers

When addressing the program and data segments through different page tables, the page table used by the program is termed the normal page table, that used for data is called the alternative page table. Before it can be used, the alternative table number must be loaded to the Paging Control Register (PCR).

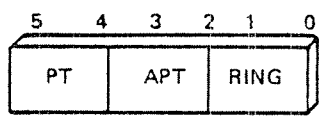


Fig. 5. Paging Control Register

There are 16 of these registers, one for each hardware interrupt level (see chapter 4). RT programs run on level 1 and all programs use the same PCR register. The operating system restores the PCR register from the RT description when one program is interrupted and another one started.

The user is not allowed to access this register himself, but must execute the monitor call ALTON (MON 33), requesting the operating system to perform the loading of the PCR. ALTON is available in Fortran as a subroutine call. The argument to ALTON is the page table number (0:3). In MAC, the A register contains the address of the parameter list; the parameter list contains the address of the parameters. ALTON has no error return - if an illegal parameter is specified, the call is ignored.

In MAC, the code for setting PT3 as alternative is

```
        ALTON=33

        LDA  (PAR       % A contains address of parameter list
        MON ALTON       % MON 33
          ...           % Next instruction to be executed

PAR,    (3              % Set PT3 as alternative Page Table
```

In Fortran, a subroutine call for performing the same operation is

```
        CALL ALTON(3)
```

To turn the alternative page table mechanism off, the monitor call ALTOF (MON 34) or Fortran subroutine call ALTOF may be used. This sets the alternative page table equal to the normal page table in the RT description. MON ALTOF has no arguments and no error return.

MAC example:

```
        ALTOF=34

        MON ALTOF           % MON 34
```

3.7.2 Bit 0 of the status register

Even if an alternative page table has been defined through ALTON, the hardware addressing mechanism does not use alternative addressing unless bit zero of the status register has been set. The Fortran routine also sets this bit, but MAC programmers should set this bit explicitly. This can be used to save time (compared to ALTON/ALTOF calls) if there is frequent switching between the ordinary and the alternative page table, e.g. if a block of data should be moved from PT1 to PT3. The code to move 100 (decimal) words from a segment addressed through PT1 to another segment addressed through PT3 would look like this in MAC

```
        ALTON=33

        LDA (PAR
        MON ALTON
        SAX -144            % set counter to -100 decimal
LOOP,   BSET ZRO            % disable alternative PT
        LDA I,X (SOURCE     % load Areg from PT1
        BSET ONE            % enable PT3
        STA I,X (DEST       % store in alternative PT
        JNC LOOP            % increment, repeat if negative
        GO SOMEWHERE
PAR,    (3
)FILL
```

The program is loaded at PT1.

The alternative page table mechanism is also available from background (timesharing) programs, but the only alternative page table allowed is PT3. If any other page table is specified, the call is ignored. RT programs using the alternative page table use two different segments in the two tables, while a two-bank background segment is treated as one segment covering both tables.

3.7.3 Monitor calls with the alternative page table mechanism on

In most cases, monitor calls work in exactly the same way when executed with the alternative page table mechanism turned on. Parameters for the monitor calls are, like other data, found in the alternative area.

If the user program turns the status register bit 0 on and off, the user should be aware that SINTRAN III, when fetching parameter values, will use the alternative page table found in the RT description set on the ALTON call, regardless of the status register bit 0 of the user program. The only way to force monitor call parameters to be taken from the normal page table is to turn off the APT mechanism through the ALTOF call.

SINTRAN III version D (Sintran 80A) and earlier:

Installations running Sintran versions prior to version E should be aware that parameter transfer is different for some monitor calls. A number of calls will fetch their parameters from the normal rather than the alternative page table, even after an ALTON has been executed and the status register bit 0 set. The user should consult Norsk Data for details. Also the new Fortran, Cobol, Planc and Pascal compilers will be unable to generate code for two-bank execution when older versions of Sintran are used.

3.8 Page faults

If the WPM, RPM and FPM bits in the page table are reset, no access to the corresponding page is legal. Attempted access generates an interrupt. This usually indicates that the page is not present in memory and must be read from disk before the location can be accessed. The term page fault is used for this condition.

This interrupt starts an operating system routine to look up the disk address of the requested page in the system tables. The routine must find an available page in physical memory and initiate a disk transfer. After the page has been copied to memory, the page table must be updated by setting the RPM, WPM and FPM bits appropriately and inserting the physical page number to which the transfer was made.

Page faults are detected and handled automatically. However, the program causing the page fault is suspended and another one started while the disk transfer takes place. When transfer is complete, the program may continue execution. The exact time for restarting it will depend on the priorities this program and of the other programs in the execution queue.

The operating system routine may find that the addressed location is not within the current segment(s). In that case there is no page on the disk corresponding to the addressed page and the program executing the reference is aborted with the error message OUTSIDE SEGMENT BOUNDS or PAGE FAULT FOR NON-DEMAND.

3.9 Swapping

If all physical memory pages are in use when space is needed for a program and its data, this space must be released by removing one of the virtual pages from memory. This process is called swapping.

If the page to be removed has been modified since it was read from disk, it must be written back before the memory page is overwritten. This is detected by testing the WIP (Written In Page) bit in the page table. If the WIP bit is set, a store operation to one of the locations in the page has been executed. If the bit is reset, no modification has been made; an exact copy of the page is present on disk and write back is not required.

3.9.1 Selection of a "victim" for swapping

The selection of a page to be removed may strongly affect both the overhead and the amount of swapping necessary. The main strategy in SINTRAN III is as follows:

All active segments are ordered in a queue (i.e. a linked list) called the segment queue. The first segment in the queue is the one that has been active most recently, the last segment is the one used least recently. The queue is updated every time a page is brought into memory for a segment.

The least recently used segment is the one which has to give up one of its pages. If any pages have their PGU bit reset, one of these is selected. If all pages have been used, then if one or more of the segment's pages has its WIP bit reset, one of these is picked at random. Otherwise any page is selected at random and written back to disk before the memory page is granted to the requesting segment.

This strategy may be modified by a number of factors: a system variable called MAXP may limit the number of pages any segment may have in memory concurrently. If the requesting segment already has this number of pages in memory, one of its own pages will be taken. (The variable MAXP can be inspected and modified by user SYSTEM through the @SINTRAN-SERVICE-PROGRAM command *CHANGE-VARIABLE. The @SINTRAN-SERVICE-PROGRAM is not available to user RT.)

If a request to fix a segment in memory has been executed, under no circumstances are pages removed from that segment.

3.10 **Process switching**

When the attention of the CPU switches from one program to the next, the page table entries must be modified to point to the pages of the (new) segment. The PCR register is updated. This is done by the operating system, invisible to the interrupted program.

The actual data pages are retained in memory; pages of one segment will not interfere with pages of another segment.

If any of the pages in the segments of the new program are already in memory, the physical page numbers are entered in the correct page table(s). The PT entries of the remaining pages are zeroed, so that later accesses to these pages causes a page fault interrupt.

3.11 **The Paging Off (POF) area**

When the paging system is turned off through the POF or the PIOF machine instruction, there is no translation of the virtual (program) address. These instructions are permitted for ring 2 or 3 programs only. The virtual address is interpreted as a physical address, referring to the lowest 64K words of physical memory.

The resident part of SINTRAN III is mapped through the lowest part of PT0 (page numbers 0 to 32) directly onto the corresponding physical addresses. Thus, resident Sintran may be accessed through PT0 with paging on or directly with paging off. The virtual and physical addresses above the Sintran resident area do not coincide. The locations addressed with paging off are not available through the page tables and are called the Paging Off (POF) area. (This is a software convention; it is not dictated by hardware.)

The POF area contains essential data and code used by the operating system. Unintended modification of these variables will almost certainly be fatal to the operating system. User RT programs should never turn off the paging system. Modification of POF area requires a detailed knowledge of the operating system.

A program may test bit 16B in the status register to determine whether the memory management system is turned on. If this bit is set, the memory management system is on.

4 THE INTERRUPT SYSTEM

External devices signal that they want the attention of the CPU by giving an interrupt - sending an electrical signal detected by the CPU. When the CPU recognizes the interrupt, it starts executing a routine to serve the device.

An interrupt may also be generated within the CPU to suspend the executing program and force another routine to be executed. E.g. if a program addresses a location on a page that is not in primary memory but swapped out on disk, hardware in the memory management system generates an internal interrupt starting a routine to bring the missing page into memory.

Different serve requests may have different degrees of urgency. Interrupts are classified into 16 levels, with level 15 as the highest (most urgent) and level 0 as the lowest (least urgent). If an interrupt on a higher level occurs, all activity on lower levels are suspended until this interrupt has been handled.

A full register set is associated with each interrupt level. This includes a program counter; consequently, 16 different programs may be ready for execution at any one time. The program counters may point to any location in any program, but on the higher interrupt levels the program is always a repetitive program set up to handle incoming interrupts on that level. The interrupt levels are also called program levels (PL0 through PL15).

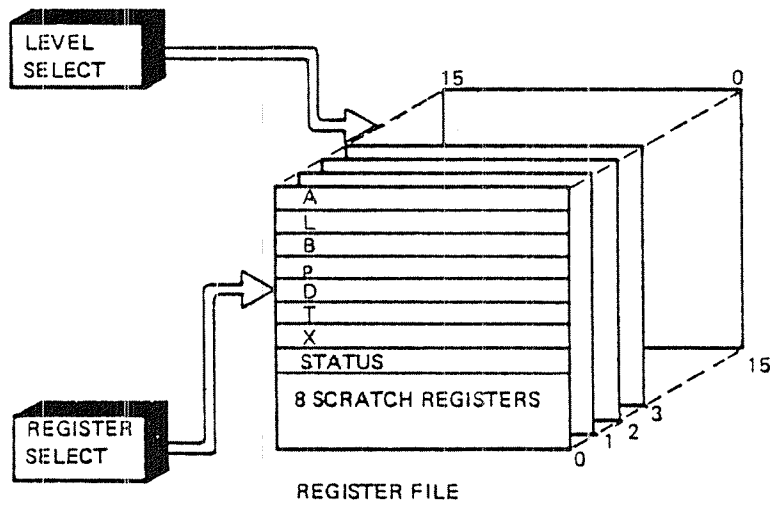


Fig. 6. The register file

When an interrupt occurs on a given level, the CPU starts executing instructions from the location indicated by the program pointer at that level. As each level has its own PCR, the program on the higher level will usually use a different page table than the interrupted program (all standard interrupt handlers use PT0) and thus be independent of it. This program continues to run until

Either; the program voluntarily gives up its right to the CPU when it has completed serving the interrupt. This happens when the WAIT machine instruction is executed. The program counter is positioned at the instruction following the WAIT and when the next interrupt (at that level) occurs, execution is resumed at this point.

Or; an interrupt at a higher level occurs. This immediately suspends activity at any lower level and start execution at the point indicated by the program counter at the higher level. The higher level will keep the CPU until it executes a WAIT or is interrupted by a level.

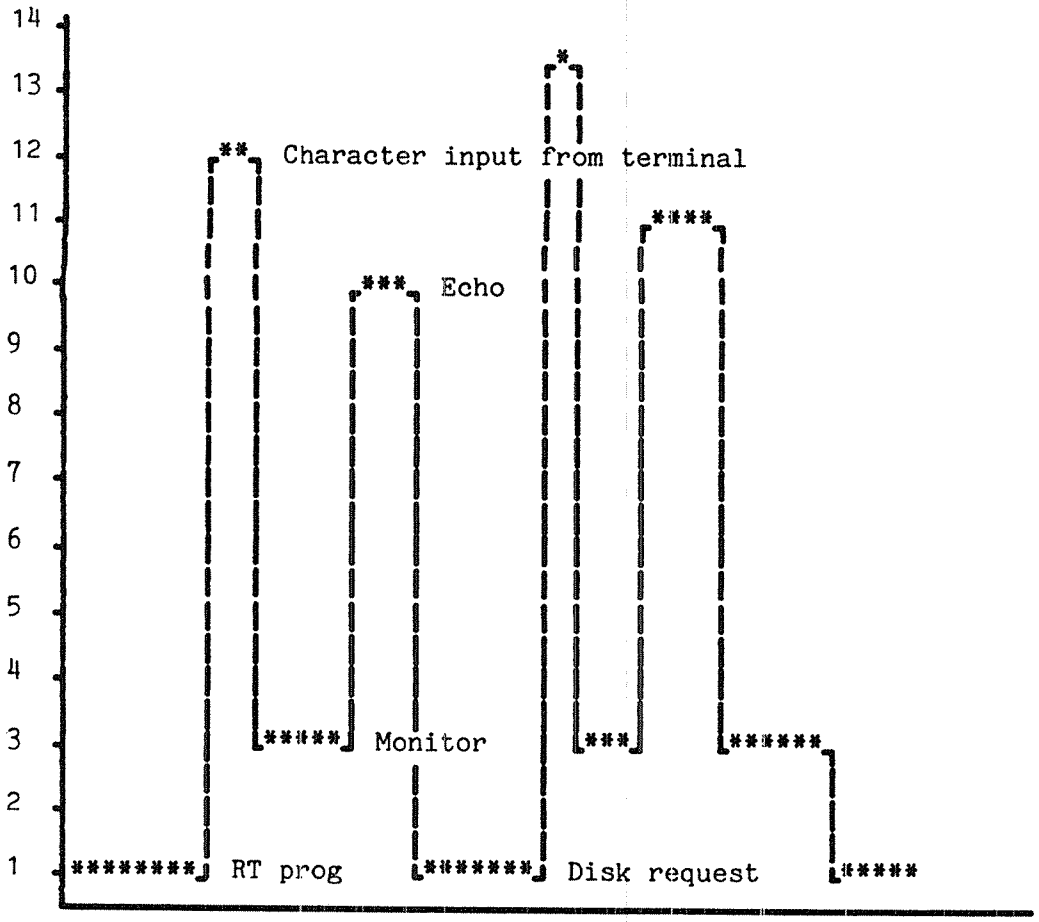


Fig. 7. Level switching

When a WAIT is executed, the execution of the lower level program is resumed wherever it was interrupted. With the exception of the time delay and modification of common data, the execution of the higher level routine is completely invisible at the lower level.

Several devices may be connected to each of the interrupt levels 10 to 13; when the program executes an IDENT instruction an identification of the interrupting device is returned. If several interrupts are generated simultaneously, they are queued and handled sequentially. Level 15 may have only one interrupt source.

An interrupt at any level may be generated by a program as well as by a hardware device or dedicated hardware in the CPU. Both software generated interrupts and interrupts generated by the CPU hardware are called internal interrupts.

4.1 Interrupt level assignments in Sintran III

The interrupt levels are used by Sintran III in the following way:

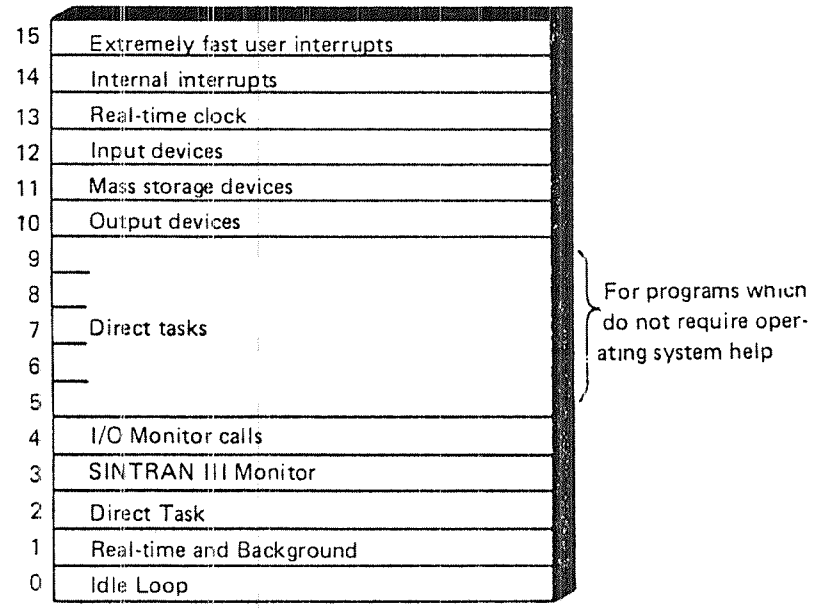


Fig. 8. Sintran interrupt level assignments

LEVEL 15 is not used by SINTRAN III but may be used by special devices requiring immediate access to the CPU. Only one device may use this interrupt level.

LEVEL 14 handles internal interrupts detected by special-purpose hardware in the CPU. This includes page faults, protect violations, illegal or privileged instructions, etc. Execution of the MON instruction also generates a level 14 interrupt.

LEVEL 13 receives an interrupt every 20 ms from a hardware clock called the Real Time clock. The routine at this level maintains a count of the number of clock interrupts received which can be divided down to larger units in order to keep track of wall clock time. HDLC input and errors in the multiport memory system are handled by the level 13 routine.

LEVEL 12 is activated by interrupts from character devices such as teletypes and screen terminals, paper tape or card readers and handles the data input from such devices. HDLC output and the ND-500 driver are handled by level 12.

LEVEL 11 controls mass storage (DMA) devices, such as disks, floppy disks and magnetic tape. Both input and output are performed at this level.

LEVEL 10 is used for character output, like terminals, printers, paper tape punches etc.

LEVEL 9 to LEVEL 6 are not used by SINTRAN III. RT programmers can write their own direct tasks (see chapter 17) running on these levels. Direct tasks are activated by a software interrupt.

LEVEL 5 is used by XMSG, an optional part of SINTRAN III. If not included level 5 is available for direct tasks. An introduction to XMSG is given in chapter 20.

LEVEL 4 handles the monitor calls INBT/OUTBT, M8INB/M8OUT, B8INB/B8OUT and B4INW. If the level 14 routine has classified a monitor call as one of these, it generates an interrupt to level 4 before giving up priority and the level 4 routine performs the actual serving.

LEVEL 3 executes the RT monitor, which includes routines for handling system queues, priorities of RT and background programs and monitor call administration. Segment administration is executed on level 3, and so are monitor calls without parameters.

LEVEL 2 is available for direct tasks.

LEVEL 1 executes all user RT programs and background programs and some less time critical routines used by the RT monitor. Most monitor calls are executed on level 1, after classification on level 3. As only one program is active on each level at a time, level 1 is switched from one program to another by forcibly setting the program counter in the register file and switching segments by setting the PCR and PT contents. This is done by the monitor on level 3.

LEVEL 0 runs an idle loop. No other routine runs on level 0 as the idle loop never gives up priority. (In fact, since there are no lower levels to give attention to, a WAIT instruction executed on level 0 is effectively ignored.)

4.2 The interrupt handlers

It is extremely important that higher level routines do not delay lower level routines unecessarily, as the delay could be too long for devices handled at lower levels to be served properly. In many cases, the higher level routines will only classify the interrupt and give an interrupt to a lower level based on this classification.

E.g. if a character input from a terminal (level 12) occurs while the CPU is busy serving a monitor call (handled on level 14), the level 14 routine must complete and also give the level 12 routine time to handle the incoming character before the next character from the terminal arrives. If the level 12 rutine has not had time to read the character buffer on the interface card before the next character

arrives, it loses the first character.

Therefore the routine on level 14 does not perform the actual serving. E.g. it may classify the monitor call as an I/O request and set up the routine running on level 4 to handle it before it executes a WAIT. The higher level character input is completely processed before level 4 is allowed to start execution.

This technique is used whenever some analysis of interrupt conditions is required to determine the urgency of an interrupt.

Because execution of the interrupt handler starts immediately when an interrupt arrives, it must be resident in memory at any time when interrupts are enabled. Also, the page tables must be properly loaded in advance (the level 3 routine handling the page tables is not allowed CPU time while a higher level interrupt is handled). To save the time spent loading the PTs handlers are usually placed in the paging off (POF) area and the paging system turned off.

4.3 Interrupt detection and programmed interrupts

An external interrupt sets a bit corresponding to the level of the handler for the device in the Priority Interrupt Detect (PID) register. This register can also be modified by a program. Setting a bit in this register causes an internal interrupt in a manner equivalent to an external interrupt. Only levels 0 to 9 can only be activated this way.

A bit set in the PID register alone does not cause the CPU to give control to the routine at that level. The Priority Interrupt Enable (PIE) register can mask out interrupts at any level and the program level active is the highest one with both the PID bit set (marking a pending interrupt) and the PIE bit set (marking that the interrupt is allowed).

The PID and PIE registers are programmed through privileged instructions allowed only in ring 2 and 3. The TRR instruction transfers the A register (on the current level) to the specified register. The MST (masked set) instruction selectively sets the bits in the specified register that are also set in the A register leaving other bits unmodified. The MCL (masked clear) instruction selectivly clears those bits set in the A register, leaving other bits unmodified.

E.g. after the level 14 routine has classified a monitor call as an I/O request to be handled at level 4, the following instructions are executed:

```
SAA 20          % Set bit 4
MST PID         % Set PID bit 4, leave others unmodified
WAIT            % Wait for next level 14 interrupt
```

Before WAIT is executed the routine has set the program counter on level 4 to one specific routine. This is done through the IRW instruction. If the routine on level 4 is called L4R21, the MAC code to start this routine is:

```
LDA (L4R21          % Load A with routine address
IRW 40 DP           % Set P-reg lev 4 to this routine
SAA 20              % Cause an interrupt on level 4
MST PID             % When level 4 starts executing,
WAIT                %   routine L4R21 is performed
```

If a programmed interrupt to a higher level is executed and the level is enabled, the executing program is stopped immediately after the MST PID instruction and not resumed until the higher level program executes a WAIT machine instruction.

If a programmed interrupt to a lower level is exectued, there is no activity on the lower level until the executing program performs a WAIT machine instruction.

4.4 Turning off the interrupt system

Critical code sequences may be sensitive to interrupts, for reasons of timing and data consistency reasons. Timing considerations occur mostly with high speed devices, data consistency in protection of shared resources.

The operating system may use the IOF or the PIOF machine instruction to turn the entire interrupt system off. Then interrupts are not honored; they are queued and executed when the interrupt is turned on by the ION or the PION instruction. A program may test bit 17B in the status register to determine whether interrupt is turned on (bit set) or off (bit reset).

User RT programs should not turn the interrupt system off.

5 THE SEGMENT FILE

All programs and directly accessible data (segments, as opposed to data in files read through the file system) are located in one or more segment files. These are contiguous files belonging to the user SYSTEM.

The segment files must be available when user RT starts using the RT loader. The creation of segment files is usually a one time operation which must be performed by the system supervisor. The only time when a segment file is created (or if possible expanded) is if the segment file overflows.

5.1 The organization of the segment file

The area occupied by the segment file is by the RT loader split in logically separate parts called segments. Each segment is an image of a part of virtual memory; the maximum segment size is (normally) 128K words.

Associated with the segment is an address giving the (logical) lower address and segment size. This information is kept in the segment table (see chapter 6).

The segment file may have areas not currently occupied by any segment; this space can be used when a new segment is created.

A segment is identified by a segment number between 0 and 376B. By convention, segment numbers are specified in octal and commands with segment number arguments expect octal input.

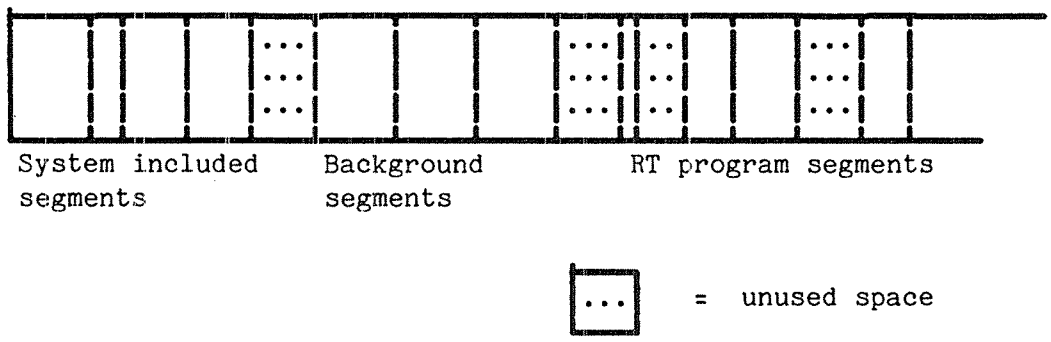


Fig. 9. Segment file

The maximum number of segments is determined at system generation time; it can be up to 376B. Default allocation of numbers to new segments chooses the first unused number when the segment is created, but a programmer can request any free segment number. A list of free segment numbers can be obtained through the RT Loader command *LIST-FREE-SEGMENTS.

Although not strictly a segment, in some commands the RTCOMMON area is indentified by segment number zero. The user should pay close attention to the meaning of segment number two (RTCOMMON or resident) in each command. Segment number zero in the segment file contains a copy of the "resident" part of the SINTRAN III operating system (pages 0:35 in PT0). This segment cannot be accessed by an RT program.

RTCOMMON is in some commands and subsystems (@LOOK-AT SEGMENT, DMAC) referred to as segment number 1.

5.1.1 The system included segments

The lowest numbered segments are called system included segments and contain parts of the operating system. These are defined at system generation time rather than built by the RT Loader.

Segment no 2B:	Memory image and POF
3B:	Command processor (OPSEG)
4B:	RT Loader
5B:	System segment for error program
6B:	File system common segment
7B:	DMAC segment
10B:	Used by RT Loader (RTFIL table)
11B:	Error log segment
12B:	Initial reentrant file system segment no. 2
13B:	Initial RT Loader (save area for RT Loader)
14B:	Error program segment
15B:	Initial service-program and MAIL segment
16B:	Initial ND-NET segment
17B:	File user data segment for RT programs
20B:	ND-500 standard domain table segment
21B:	ND-500 table segment (name segment)
22B:	Reentrant file user segment no. 1
23B:	SINTRAN-SERVICE-PROGRAM and MAIL
24B:	Reentrant file system segment no. 1
25B:	Reentrant file system segment no. 2
26B:	Reentrant file user segment no. 2
27B:	ND-NET segment
30B:	ND-500 System Monitor segment no. 1
31B:	ND-500 System Monitor segment no. 2
32B:	RT accounting segment
33B:	XMSG segment (POF)
34B:	XMSG segment (XROUT)
35B:	Reserved for XMSG
36B:	TADADM segment
37B:	RT Loader data segment

Segments not used by a particular configuration are empty. Unused segments with numbers lower than the highest system included segment are not available. The list applies only to the I version of SINTRAN III.

5.1.2 The background segments

Two segments are used for each background/batch process in the system; these usually follow the standard system segments in the number sequence. RT programmers need not be concerned about these segments, but must not clear or modify them.

Each background/batch process has a 128Kword segment used as a swapping area. Whenever a page belonging to a background process has to be copied back to disk because the space in physical memory is needed, the corresponding page in the background segment is used.

In addition, there is a 5K segment called the system segment (not to be confused with the system included segments) used for a number of non-reentrant routines in the operating system and as a data area for the background processor and the file system. This segment is located at page table 0 from address 22000B to 33777B, while the background segment is located at page table 2 (and if a two-bank system also page table 3).

A background "two bank system", using the alternative page table mechanism, may occupy up to 128K of virtual memory. A background segment of 128K must be available. By default are all background programs in SINTRAN III H version "two bank systems". The SINTRAN III command:

```
@CHANGE-BACKGROUND-SEGMENT-SIZE <term.no> 128
```

will deallocate the 64K segment for the specified <term.no> and allocate a segment of size 128K. This command is permitted only for user SYSTEM. The same command may be used to later reduce the size to 64K if required.

5.2 The contents of a segment

In principle, a segment may contain any binary information that will fit within its limits, without regard to storage format. Thus, it may contain binary or symbolic data, instructions or a combination of instructions and data.

A segment may contain the instructions for one or several RT programs. Putting several RT programs onto one segment means these programs share data locations; they may share routines or even the entire program code. If several programs share data and are also active concurrently, special consideration must be given to reentrancy, as modification of a location by one program has immediate effect for the other programs using it. These problems are discussed in chapter 14.

5.3 The use of a segment

When an RT program on a segment is activated, the required part of that segment is automatically copied to memory. The extent of the required part depends on the segment: a demand segment is copied one page at a time as the pages are needed, a nondemand segment is copied in its entirety into memory as soon as any of its pages are needed.

Demand/nondemand is defined in the *NEW-SEGMENT command by the parameter <segment type>. The value ND indicates NonDemand, DM indicates DeMand. The default value is ND. To declare segment 270 as a demand segment, it must be allocated as follows (using default values for the remaining parameters):

```
*NEW-SEGMENT 270,,
SEGMENT TYPE: DM,,,
*
```

Copying of a segment to memory is invisible to the RT program (with the exception of the time consumed). Before the segment is removed from memory, modified pages are written back. From a logical point of view the segment may be treated as if modifications were made directly to the segment file rather than to a copy in memory.

Monitor calls are available to force premature write-back of a segment to the segment file, or to place a segment in memory without accessing any location.

An entire segment can also be placed in memory through a command or a monitor call and remain there until explicitly released. This is called fixing a segment. Fixing is not a property of the segment; it is never done automatically except by a command or monitor call.

Demand/nondemand and fixing of segments are discussed in detail in chapter 8.

5.4 Creating a segment

A new segment is allocated by the RT Loader command *NEW-SEGMENT. The size of the segment is automatically determined at load time. Until the end of the load to the new segment is indicated by the *END-LOAD command, loading takes place to the scratch file ("file number 100"). As soon as *END-LOAD is typed, the RT loader searches the segment file(s) for an free, contiguous area large enough to hold the loaded code and data. This determines the size of the segment.

The segments are not necessarily ordered according to number on the segment files. If a segment is deleted and a new, larger one allocated with the same segment number, it will not usually fit into the same position in the segment file. New segments are placed in the last file specified in a *SET-SEGMENT-FILE command, see section 5.8.1

5.5 The segment file bit map

The user may inspect the bit maps for segment files by using the RT loader command *DUMP-SEGFILE-BITMAP. Parameters are segment file number (from 0 to 3) and output file. Defaults are all four segment files and output to the terminal. A bit which is set in the bit map indicates that the page is occupied, a reset bit that the page is available. The number of free pages and the largest contiguous free area are reported (see example in the next section). The latter may limit the maximum size of new segments.

It is not necessary to use this command during loading, but the system supervisor may want to inspect the bit map in order to evaluate the amount of space wasted in the segment files. If there are many small, non-contiguous free areas is high, reorganizing the segment file may increase the maximum size of new segments.

5.6 Reorganizing the segment file

If there are many small, non-contiguous free areas in the segment file, it can can be compacted by moving the free areas to the upper end. This is done through the RT loader command *REORGANIZE-SEGMENT-FILE, with the segment file number (0 to 3) as parameter. If a segment file number is not specified, all segment files are reorganized, but segments are not moved from one file to another.

System included segments listed in section 5.1.1 are not moved by this command.

When this command is performed, no segment should be in use by RT programs and no reentrant segments (defined by the Sintran @DUMP-REENTRANT command) should be in use by a background process.

Example (the compacting is illustrated by inspecting the bit map of the segment file):

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*DUMP-SEG-BITMAP 0,,

    0 177777 177777 177777 177777 177777 177777 177777 177777
  200 177777 177777 177777 177777 177777 177777 177777 177777
  400 177777 177777 177777 177777 177777 177777 177777 177777
  600 177777 177777 177777 177777 177777 177777 177777 177740
 1000 000000 000000 000000 000036 000000 000000 000000 000001
 1200 160000 000000 000000 000000 017000 000000 000000 000000
         1 160000 000000 000000 000000 017000 000000 000000
                                000000 000036 000000 000000
15400 177777 17777                          0000 000000 000000
15600 177777 177777 177777 177777                       00000
16000 177777 177777 177777 177777 177777 17777
16200 177777 177777 177777 177777 177777 177777 17
16400 177777 177777 177777 177777 177777 177777 177777 177777
16600 177777 177777 177770 000000 000000 007700 000000 000000
17000 000000 000000 001700 000000 000000 177600 000000 000000

FREE PAGES ON SEGMENT FILE: 4523
NUMBER OF CONTINUOUS FREE PAGES:  723
```

```
*REORGANIZE-SEGM-FILE 0
*DUMP-SEGF-BITMAP 0,,

    0 177777 177777 177777 177777 177777 177777 177777 177777
  200 177777 177777 177777 177777 177777 177777 177777 177777
  400 177777 177777 177777 177777 177777 177777 177777 177777
  600 177777 177777 177777 177777 177777 177777 177777 177777
 1000 177777 177777 177777 160037 177777 177777 177777 177761
 1200 177777 177777 177777 170000 017777 177777 177777 177400
           177777 177777 177777 140000 017777 177777 177777
                                177000 000037 177777 177777
15200 177777 17                    777 000000 017777 177777
15400 177777 177777 177777 177
15600 177777 177777 177777 177777 17
16000 177777 177777 177777 177777 177777 177777 1
16200 177777 177777 177777 177777 177777 177777 177777 177777
16400 177777 177777 177777 170000 000000 000000 000000 000000
16600 000000 000000 000000 000000 000000 000000 000000 000000
17000 000000 000000 000000 000000 000000 000000 000000 000000

FREE PAGES ON SEGMENT FILE: 4523
NUMBER OF CONTINUOUS FREE PAGES: 1236
*EXIT
```

5.7 Creating a new segment file

If reorganizing the segment file is not sufficient to make room for new segments, a new segment file must be created. In rare cases, the disk pages immediately following the existing segment file(s) are available and the file can be extended with the Sintran command @EXPAND-FILE, but usually a new file must be created.

The total size of all segment files cannot exceed 40000B pages.

5.8 Several segment files

If SEGFIL0:DATA overflows, another one can be created called SEGFIL1:DATA. (The next one after that is called SEGFIL2:DATA and the last one allowed is SEGFIL3:DATA). These may be placed on any directory that is not on a sub-unit.

Segment files must be located at disk pages with physical address below 177777B. Contiguous free space can be found by the Sintran command @DUMP-BIT-FILE (see the Sintran Reference Manual, ND-60.128).

When the file has been created it must be declared as a segment file by the @SINTRAN-SERVICE-PROGRAM command *DEFINE-SEGMENT-FILE. Users SYSTEM and RT must have read and write access.

As the segment files belong to user SYSTEM, user RT cannot execute these commands. Creating a segment file is a one time operation and as soon as SYSTEM has created the file and defined it as a segment file, it is available for use by user RT. User RT must have read and write access (RW) to the file. If there are several main directories in the system, user RT and SYSTEM should be in the same one and should not have space in any other main directory.

SYSTEM must create the segment files as contiguous files by the Sintran command @ALLOCATE-FILE. In most systems, one segment file is created at installation time, with sufficient space for ordinary use.

5.8.1 Selecting the file to be used for new segments

When one segment file is full, new segments must be allocated in another file. The file to be used is specified through the command *SET-SEGMENT-FILE:

```
*SET-SEGMENT-FILE
SEGMENT FILE NO.:1
*
```

The segment file is specified by the index given in the *DEFINE-SEGMENT-FILE in the @SINTRAN-SERVICE-PROGRAM and is in the range 0 to 3.

The specified file is used for all segments created until a new *SET-SEGMENT-FILE command is given. It is not reset when the user leaves the RT loader. When the command is given, the segment file must be defined, but no warning is given if the file is later deleted as a segment file. In that case, an attempt to create a new segment returns an error message: SEGMENT FILE NO. n IS FILLED (n is either 0,1,2 or 3) when the *END-LOAD command is executed.

The segment file number is not determined until *END-LOAD is executed. If the *SET-SEGMENT-FILE command is issued after a *NEW-SEGMENT command, but before an *END-LOAD command, the segment is allocated in the newly specified file.

5.8.2 Name

The standard names of the segment files are SEGFIL0:DATA, SEGFIL1:DATA, SEGFIL2:DATA and SEGFIL3:DATA. Other names are legal, but use of standard names is recommended. The maximum number of segment files is four. SEGFIL0:DATA must be located in the main directory (if there is more than one main directory, user SYSTEM should not have space in more than one) but the other three may be in any directory that is not located on a sub-unit.

5.8.3 Location

No page of a segment file may have a higher page number on the disk than 177777B. For this reason, if a segment file is to be created on the directory, this should be done as soon as possible after the directory is created, before the disk has user files that may occupy part of the required area.

On disk drives split into sub-units, a segment file may not be located on a subunit $\neq 0$. This applies e.g. to a 150 Mb disk treated as two 75 Mb directories or a 90 Mb (6 times 15 Mb).

The SINTRAN III command @CREATE-FILE does not ask for a disk address, but places the file in any sufficiently large contiguous area. The @ALLOCATE-FILE command should be used to specify the location.

5.8.4 Size

The RT loader uses a bit map to indicate free and available pages. The total size of this bit map is 40000B (16384) bits, limiting the total size of all segment files.

The size required for the segment file varies considerably from one system to another. SEGFIL0:DATA should be large enough to hold the segments required by Sintran, from 300 to 1000 (octal) pages depending on the configuration and the number of background segments. The number of pages used by the system is printed on the console during a "cold start" (MACM/HENT), assuming 128 page background segments.

Each terminal requires 128 pages ("two-bank systems") plus a 5 page "system segment" (used by the command processor and a few non-reentrant I/O routines). If "one-bank" systems are to be run on the terminals, the terminal segment size migth be 64 pages plus the 5 page system segment.

Reentrant subsystems (listed by the Sintran command @LIST-REENTRANT) also require space on the segment files roughly equal to the size of the :BPUN files.

All of these occupy space unavailable to RT programmers; the space required for RT programs is additional to the above. If RT activity varies much over time, some space is lost because the holes between existing segments are not big enough to hold a large new segment.

An example:

> A moderately sized system requires 320 (500 octal) pages for the system segments (excluding background segments). There are 12 terminals, all running two bank systems, for a total of (128 + 5) * 12 = 1596 pages. The sum of the sizes of compilers, editors, utility systems like NOTIS and the Sort-Merge package is approximately 500 pages. On average, ten RT processes with an average of 40 pages each for code and data brings the minimum size of the segment file to around 3000 pages. Depending on the required safety margin, the size of the segment file(s) should be between 3000 and 3500 pages.

These pages may be in one or more segment files. If SEGFIL0:DATA is smaller than this, a new segment file must be created.

5.8.5 Summary

To create a new segment file:

1. Determine the amount of extra space needed
2. Log in as user SYSTEM
3. Find a big enough free are on page address less than 177777B by @DUMP-BIT-FILE
4. Create a file with @ALLOCATE-FILE
5. Define the file as a segment management file by the *DEFINE-SEGMENT-FILE
6. Ensure that RT and SYSTEM have RW access to the file
7. Set the segment file number to be used in the RT loader

Example:

Implementation of the Sibas database system requires 100 new pages for Sibas itself. Sibas RT application programs are expected to fill approximately 300 pages and other RT programs at least 300 pages more than currently available. A 750 page segment file should be made on directory PACK-TWO:

```
ENTER SYSTEM
PASSWORD:
OK
@DUMP-BIT-FILE PACK-TWO
BLOCK NO: 0,,

000000  000001 000000 000000 000000 000000 000000 000000 000000
000010  000000 000000 000000 000000 000000 000000 000000 000000

@DU-BIT-FI P-TWO 1,,

000000  000000 000000 000000 000000 000000 000000 000000 000000
000010  000000 000000 000000 000000 000000 000000 000000 000000

@DU-BIT-FI P-TWO 2,,

000000  000000 000000 000000 000000 000000 000000 000000 000000
000010  000000 000000 000000 000000 000000 000000 000000 000000
```

(There is an unused 750 page area from page 20B on:)

```
@ALLOCATE-FILE
FILE NAME: (PACK-TWO:)SEGFIL1:DATA
PAGE ADDRESS: 20
NUMBER OF PAGES: 750
@SINTRAN-SERVICE-PROGRAM
#DEFINE-SEGMENT-FILE
MEMORY? Y
SAVE-AREA? Y
SEGMENT FILE NO.: 1
SEGMENT FILE NAME: (PACK-TWO:)SEGFIL1:DATA
#EXIT
@RT-LOADER

REAL-TIME LOADER, SINTRAN III - VERSION G

#SET-SEGMENT-FILE
SEGMENT FILE NO.: 1
#EXIT
```

6 SYSTEM TABLES AND QUEUES

This chapter is not required reading by users who simply want to load and execute a small program with no special requirements. Readers unfamiliar with RT programming in general may want to skip it during their first reading of this manual.

The RT monitor keeps descriptions of all RT programs and system resources in various tables and queues. These are normally invisible to programmers, and knowledge of their formats is not required to write RT programs.

However, even a moderately advanced programmer must be familiar with the concept of the RT description and to some degree the datafields. Familiarity with the queues is required to determine why a system does not perform as expected or goes into a deadlock. The section on segment handling is background information about how SINTRAN III uses the memory management system hardware to implement the facilities available in monitor calls. Programmers have no direct influence over segment administration.

RT programs and system resources are administered by the RT monitor. The RT monitor runs on interrupt level 3 (see chapter 3) and is thus independent of user RT programs.

6.1 Program management

Sintran handles RT descriptions, describing programs and datafields, describing devices. These are elements in a number of queues (linked lists) handled by the RT monitor.

There is no descriptor of data visible to programmers. Although the RT monitor handles data elements in the form of segments, the descriptors of the segments are not used by RT programs except indirectly through certain monitor calls and their properties are hidden.

6.2 Queue elements

RT descriptions and datafields appear in five queues:

Queue:	Queue elements:	Queue head:
Monitor queue	Datafields	MQUEUE
Execution queue	RT description	BEXQU
Time queue	RT description	BTIMQU
Reservation queue	Datafield	RT description
Waiting queue	RT description	Datafield

6.2.1 The RT description

Several RT programs can run concurrently. An RT program is active, suspended or passive. It cannot be multiply active but may be executed repeatedly. If an execution is requested when the program is already active, it is set up for reexecution as soon as it ends. Several requests to execute while it is active only causes one reexecution. Repetitive execution at regular intervals can be specified.

Each RT program is described by an RT description which contains the information required by the RT monitor to allocate CPU time and other computer resources. The RT description represents the program in the queues and tables, i.e., a program is in a queue if its RT description is in the linked list forming the queue.

The RT description has a size of 31 (37B) words. The most accessed data (15 words) are kept in resident memory, while the less accessed are kept in the paging off area (POF). The layout is as follows:

```
 0B:      TLINK          Time queue link location
 1B: STATUS/PRIORITY     Flags/Programmed priority
 2B:      DTIM1          Absolute activation time if
 3B:      DTIM2            scheduled for later execution
 4B:      DTIN1          Interval between executions
 5B:      DTIN2            if periodic program
 6B:      STADR          Start address of program
 7B:   SEGM1/SEGM2       One or two initial segments
10B:      WLINK          Waiting queue link location
11B: ACTSEG1/ACTSEG2     One or two active segments
12B:      ACTPRI         Page table, protection ring, flags, interrupt
                           level (PCR register).
13B:      BRESLINK       Start of reservation queue
14B:      RSEGM          Reentrant segment used
15B:      WINDOW         Bit 0-7: Logical page address of the first
                           user - address used in file transfer.
                         Bit 8-15: Physical page address of device
                           buffer used in file transfer.
16B:      RTDLGADDR      Address of RT description part residing in POF
bl;
```

The part of the RT description in the POF area contains the values of the register block, and a bitmap for the reentrant segment. The first 0 - 7B locations, containing registers, are called DPREG, DXREG, DTREG, DAREG, DDREG, DLREG, DSREG and DBREG.

The locations 10 - 17B are called BITMAP, BITM1, BITM2, BITM3, BITM4, BITM5, BITM6 and BITM7. The 8 words are the 128 bits of the bitmap. Each bit indicates if a page of a reentrant segment should be fetched from a private copy (ie., the page is modified) or fetched from the reentrant segment (ie., page not modified).

The RT descriptions are stored in the RT description table. The size of this table is determined at system generation time and limits the number of RT programs in the system. An RT description is identified by the address of the first word of its entry in the memory resident part of the RT description table.

The command processor and RT Loader use the table to translate a symbolic name to an address. The RT description address is of practical use when following a chain of descriptions, e.g., during debugging or after a system crash.

Some monitor calls allow the calling program to use itself as an argument by specifying RT description address zero. Default parameter in commands is the background program controlling the terminal from which the command was executed. In all commands allocating a new descriptor, default action is to use the first free entry in the table and in most other commands the name of the RT program described is used.

A list of free RT description table entries is obtained through the RT loader command *LIST-FREE-RT-DESCRIPTIONS. The description address of an existing RT program is obtained through *WRITE-PROGRAMS, which lists all program names, their description address and initial segment numbers (word 7B in the RT description, right byte first):

```
@RT-LOADER

REAL-TIME LOADER, SINTRAN III - VERSION G
*LIST-FREE-RT-DESCRIPTIONS
OUTPUT FILE: TERMINAL

  40607  40641  40673  40725  40757  41011  41043  41075
  41127  41161  41213  41245  41277  41331  41363  41415
  41447  41501  41533  41565  41617  41651  41703  41735
  41767  42021  42053  42105  42137  42171

*WRITE-PROGRAMS
OUTPUT FILE: TERMINAL

 DUMMY    35153    0    0
 STSIN    35205    3    5
 RTERR    35237   14    5
 RTSLI    35271    0    0
 RWRT1    35407    6    0
 RWRT2    35441    6    0
 FDRT1    35473    0    0
 RWRT9    35525    6    0
 FIXRT    35557    0    0
 DUMM2    35611    0    0
 SPRT1    35643   25    0
 BAK01    36013    3   60
 BAK09    36045    3   62
 BAK10    36077    3   64
 BAK11    36131    3   66
 BAK12    36163    3   70
 BAK13    36215    3   72
 BAK14    36247    3   74
 BAK15    36301    3   76
 BCH01    40065    3  202
 TIMRT    40151    0    0

*
```

A list of the RT programs in the system may also be obtained through the Sintran command @LIST-RT-PROGRAMS. This returns some additional information: for each program the priority, the state of the program, the current program counter, the time left and the interval in case the program is periodic, and the actual segment numbers (word 21B in the RT description, left byte first):

```
@LIST-RT-PROGRAMS
OUTPUT FILE: TERMINAL
  NAME  RT-DESC  PRIOR STATUS   P-REG T.LEFT INTERV   ACTUAL SEGM

  DUMMY  35153      0 READY     3136                      0    0
  STSIN  35205      0 PASSIVE 147507                      5    3
  RTERR  35237     64 IO-WAIT 107425                      5   14
  RTSLI  35271    128 PASSIVE  12375      0      0        0    0
  ACCRT  35323     64 PASSIVE      0                      0    0
  RWRT1  35407     64 PASSIVE   6112                      0    6
  RWRT2  35441     64 PASSIVE   6112                      5    6
  FDRT1  35473     64 PASSIVE  53007                      0    0
  RWRT9  35525     64 PASSIVE      0                      0    0
  FIXRT  35557    104 PASSIVE      0                      0    0
  DUMM2  35611      0 READY        0                      0    0
  SPRT1  35643     44 W-QUEUE  47173                      0   36
  BAK01  36013     16 PASSIVE 113161                     60    3
  BAK09  36045     16 PASSIVE 102140                     62    3
  BAK10  36077     16 IO-WAIT  21401                     64   65
  BAK11  36131     16 IO-WAIT 132747                     66    3
  BAK12  36163     16 PASSIVE 113161                     70    3
  BAK13  36215     48 READY     7727                     72    3
  BAK14  36247     16 IO-WAIT 132747                     74    3
  BAK15  36301     48 IO-WAIT  21401                     76   77
  BCH01  40065     16 IO-WAIT 101330                    202    3
  TIMRT  40151     64 PASSIVE   5320      0      1        0    0
```

6.2.2 The RT name

The alphanumeric identification of an RT program may have up to 7 characters (A to Z uppercase and 0 to 9).

If a source program has a longer name, most high level languages use the first seven characters name when identifying external symbols, while MAC uses the last five characters. NORD-PL uses in the translation to MAC the first five characters.

Unlike Sintran commands and file names, RT names cannot be abbreviated.

6.2.3 Translating a name into an RT description address

An RT name may be translated into an RT description address by the monitor call GRTDA (MON 151). The program name is specified as a character string, terminated by an apostrophe and the RT description address is returned in the A register. If the program name is not known, -1 is returned.

```
        GRTDA=151

        LDA (ADRNM  % A reg = address of parameter list
        MON GRTDA   % MON 151
        STA RTADR   % Store the RT description address

ADRNAM, NAME
NAME,   'PRO2'
        )FILL
```

In the corresponding Fortran function the user must remember to supply the terminating apostrophe:

```
        INTEGER RTADR,GRTDA

        RTADR = GRTDA(5HPRO2')
```

The RT description address returned by this call is used to identify the program in all routine calls where the program is used as a parameter. As a special case, a program can obtain its own RT description address through GETRT (MON 30). Fortran syntax requires that a dummy argument be supplied. The RT description address is returned in the A register, or in Fortran as a function value:

MAC:

```
        GETRT=30

        MON GETRT
        STA RTADR
```

Fortran:

```
        RTADR = GETRT(0)
```

6.2.4 Translating an RT description address into a name

The name of a RT program, with a given RT description, can be obtained by a monitor call or a command:

```
Sintran command: @GET-RT-NAME <RT address>
Monitor call:    MON 152  % GRTNA
```

Command parameter:

<RT address> - the RT description address of an existing program

The program name is written on the terminal. A program may call MON GRTNA with the A register pointing to the parameter list, which consists of the RT description address. The name is returned in the T, A and D registers in a packed format, 6 bits per character. The name is right justified and space filled. The compression to 6 bits per character is performed by "folding" the ASCII character set, ignoring bit 6. The ASCII codes from 100B to 140B are transformed to

0B to 40B; the range 41B to 77B is unmodified. The uppermost 6 bits in the T register are unused.

```
        GRTNA=152

        LDA (RTADR
        MON GRTNA           % MON 152, name packed in TAD
        STF PCKD
        COPY ST DA
        SAX 0
        SHD SHR 4 ZIN
        JPL UPCK            % Store first
        LDD PCKD            % Fetch TA to AD
        AND (17
        SHD 2
        JPL UPCK            % Store second
        SHD 6
        JPL UPCK            % Store third
        SHD 6
        JPL UPCK            % Store fourth
        LDD PCKD+1          % Fetch AD registers
        AND (3
        SHD 4
        JPL UPCK            % Store fifth
        SHD 6
        JPL UPCK            % Store sixth
        SHD 6
        JPL UPCK            % Store seventh
        SAA 47
        JPL UPCK            % Terminate name with '
        . . .

UPCK,   JAZ NOCHAR
        SAT 40
        SKP IF SA LT DT
        AAA 100
        AAX 1
        LDT UNPCK
        SBYT        % Store third
NOCHAR, EXIT

RTADR,  (PROGR
        PCKD,0;0;0
        )FILL
```

A program may fetch its own name by giving RT description address zero. If no RT description with the specified address exists, or the program is unnamed, the TAD registers are zero. The D register should be checked, as a short name (one or two characters) may leave both the T and A registers with a zero value.

GRTNA is not available in the Fortran library. An assembler routine must be written to use the function from a Fortran program.

6.2.5 Reading the RT description

The most important data in the RT description, including the entire reservation queue of the program, can be displayed symbolically by the SINTRAN III command

```
@LIST-RT-DESCRIPTION <RT name>
```

(RT name may also be an RT description address). This command is available to all users. The segment numbers are listed with the leftmost byte first.

```
@LIST-RT-DESCRIPTION BAK13
 RING:2 PRIORITY:     48
LAST STARTED:     1 MINS   29 SECS
START ADDRESS:  76055, SEGMENTS:     77      3
P=   17507
X=   46773
T=    1100
A=   40010
D=       0
L=   17402
S=      41
B=   70047
READY
ACTUAL SEGM.:     77      3 BACKGROUND
REENTRANT SEGMENT:    211
RESERVED DATAFIELDS:
  23201
  23126

@
```

The complete RT description may be listed by user SYSTEM by the @SINTRAN-SERVICE-PROGRAM command *DUMP-RT-DESCRIPTION:

```
@SINTR-SERVICE-PROGRAM
*DUMP-RT-DESCRIPTION BAK13
MEMORY, IMAGE, SAVE-AREA OR SEGMENT? M
OUTPUT FILE: TERMINAL

TLINK:          0
STATUS:      1100
DTIM1:          0
DTIM2:          0
DTIN1:          0
DTIN2:          0
STADR:      24003
SEGM:       37003
WLINK:          0
ACTSEG:         0
ACTPRI:    100000
BRESLINK:   44250
RSEGM:          0
WINDOW:       126
RTDLGADD:  146450
DPREG:      17507
DXREG:      46773
DTREG:       1100
DAREG:      40010
DDREG:          0
DLREG:      17402
DSREG:         41
DBREG:      70047
BITMAP:         0
BITM1:          0
BITM2:          0
BITM3:          0
BITM4:          0
BITM5:          0
BITM6:          0
BITM7:          0
*EXIT
```

The @SINTRAN-SERVICE-PROGRAM is not available to user RT.

An RT program may read the RT description of any program, including its own, by the monitor call RTDSC (MON 27). The argument list is pointed to by the A register and contains the RT description address and the address of a 26 element integer array where the RT description is returned.

MON RTDSC does have an error return; if no RT description is available as specified in the first argument, -1 is returned in the A register and return is to the first location following the monitor call. If no error occurs, execution continues at the second location following the call and the number of devices connected to the program through CONCT (see section 9.3.1) is returned in the A register.

```
        RTDSC=27

        LDA (ARG
        MON RTDSC
        JMP ERROR       % Error return - go to error handler
        STA NODEV       % Save number of devices connected

ARG,    (PRO            % Dump description of program PRO
        RTDES           % Address of 26 element array

RTDES,                  % Space for RT description

RTDES+32/               % Skip 32B (= 26 decimal) locations
```

6.2.6 The segments of an RT program

The RT description has room for two segment numbers in word 7; the upper and lower bytes are denoted SEGM1 and SEGM2. These are the segments used initially when the program is initated. One or both of these may be exchanged with another segment through the monitor calls MCALL or MEXIT. The use of these calls is described in chapter 14.

The segments currently in use, whether initial or fetched by MCALL/MEXIT, are kept in the ACTSEG1 and ACTSEG2 locations.

If only one segment is in use the other segment number is zero. (This is not interpreted as segment number zero, resident memory, or as RTCOMMON but is treated as a special case by the RT monitor.)

The segment numbers of a program are listed by the RT loader command *WRITE-PROGRAMS.

6.2.7 The various fields of the RT description

TLINK is the link location used for the time queue. Because this queue uses its own link location, it is independent of the other queues in which the RT description may be linked through the WLINK location.

STATUS contains the ring and 5 flags:
- bits 10B,11B: Initial ring. See also the ACTPRI word.
- bit 13B: 5ABS, scheduled at an absolute time
- bit 14B: 5INT, periodic program
- bit 15B: 5RWAIT, in RTWAIT
- bit 16B: 5REP, reexecute after termination
- bit 17B: 5IOWT, waiting for I/O transfer

PRIORITY (bits 0:7) is the user assigned priority (programmed or through command)

DTIM1/2 internal time when program is scheduled for execution, in basic time units since system start.

DTIN1/2 interval of periodic program in basic time units

STADR initial start address (main entry point)

SEGM1/2 initial segments. A value of zero in either half indicates that the program uses only one segment. SEGM1,2 is not modified if one or both segments are replaced during execution.

WLINK link location for waiting queues or execution queue. Because there is only one link location for all queues, the program may only be in one queue at a time. The time queue has its own link location, so a program may be in the time queue and another queue at the same time.

ACTSEG the segments in use at the moment. When the program is started, ACTSEG is equal to SEGM1/2, but the initial segments may be replaced with others through monitor calls (see chapter 14). Like in the SEGM1/2 location, a value of zero in either half indicates the program has only one segment.

ACTPRI actual ring, page table, level, PTs and flags

bits	0:1	Actual ring. Equal to the ring in the STATUS word.
bit	2	Always 0.
bits	3:6	Interrupt level on which the program runs; 1 for all user RT programs.
bit	7:10B	Alternative page table, set by ALTON. Initially, when program is loaded, equal to the normal page table.
bit	11B:13B	Normal page table, i.e. the page table of the segment of the currently executing instruction.
bit	14B:15B	Initial page table, i.e. the page table of the code segment.
bit	16B	5RTOFF, inhibit starting of this program.
bit	17B	5BACKGR, background program.

BRESLINK head of reservation queue containing datafields reserved by this program

RSEGM segment number of reentrant segment, if any, otherwise zero.

WINDOW is used by the I/O system for transferring data from a device buffer to the program's data area. It does not concern the RT programmer.

RTDLGADR address of the RT description part residing in POF.

D?REG one location for each register in the register block. When program execution is interrupted by a higher priority program, while waiting for an I/O transfer, or if HOLD or RTWT are being used, all its register contents are saved.

BITMAP and BITM1-8 is a 128 bit wide bitmap, one bit for each logical page, used in a reentrant segment. If the bit is reset, the corresponding page used is in a reentrant segment which may be used by a number of programs concurrently. If the bit is set, the private copy of the page in one of the program's own segments is used.

6.2.8 Modifying the RT description

Some fields in the RT description are indirectly modified by the RT program itself or another program, through monitor calls affecting e.g. priority or segments. A few fields contain static information that is not normally modifyable.

User SYSTEM may modify any word in an RT description in octal format through the SINTRAN III command @LOOK-AT RESIDENT. The user must calculate the address from the RT description address and the displacement. This is highly dangerous and strongly discouraged. Modifications are effective until the next restart ("warm" start).

A few subfields can be modified through the RT Loader:

```
#CHANGE-RT-DESCRIPTION
RT-PROGRAM: MAINP
PRIORITY: 32
SEGMENT ONE: 235
SEGMENT TWO:
START ADDRESS:
RING:
INITIAL PAGE TABLE:
ALTERNATIVE PAGE TABLE:
#
```

This command can modify the locations PRIORITY, STADR, SEGM1, SEGM2 and ACTPRI. Default is no modification. The segment numbers must be existing segments, but there is no check to ensure that the start address is within the segment(s) specified.

6.3 Datafields

Devices are described in datafields whose size and format depends on the device. All datafields have 7 standard locations whose layout and addressing that are independent of device type. The datafield is identified by the address of the first standard location (displacement 0) and they have displacements 0 to 6 from this address. The displacements of device dependent fields are negative or above 6.

The device independent fields are:

RESLINK	Link of reserved resources
RTRES	Reserving RT program
BWLINK	Head of waiting queue
TYPRING	Device type and ring
ISTATE	State of the device
MLINK	Monitor queue link
MFUNC	Monitor level function address

All devices reserved by an RT program are linked through the RESLINK location. This chain is headed by the BRESLINK in the RT description.

RTRES is the RT description address of the program currently reserving the device. If RTRES is zero the device is free for use.

The **BWLINK** location is the head of the link of RT programs that have requested, but not been granted, access to the device.

The **TYPRING** word contains in bits 0 and 1 the lowest ring of programs to reserve the device. If a program is running in a lower ring than indicated by the TYPRING, a request to reserve the device is not honored. The uppermost 14 bits contain flags describing the device.

The contents of **ISTATE** are to some extent device dependent, but in general 0 indicates idle, 1 indicates busy, -1 indicates "no wait" mode.

MLINK is the link location for the monitor queue.

MFUNC is the address to a monitor routine to be started after the datafield has been removed from the monitor queue.

There is no commands available to display the entire datafield of a device, and a program cannot read the datafield directly. User SYSTEM can use the @SINTRAN-SERVICE-PROGRAM command *CHANGE-DATAFIELD to inspect the various subfields, identified by symbolic names. The symbolic names of the device dependent subfields are given Sintran System Documentation, Appendix A - Data Fields. (This appendix is delivered separately as publication no. ND-60.112.)

User SYSTEM can also use the @LOOK-AT RESIDENT command to inspect any data field. This is the only way to inspect datafields with no corresponding unit number. Subfields must be addressed numerically.

The @SINTRAN-SERVICE-PROGRAM and @LOOK-AT RESIDENT are not available to user RT.

6.3.1 Modifying locations in the datafield

User RT has no means of directly modifying locations in any datafield.

User SYSTEM may use the @LOOK-AT RESIDENT command to modify any word in a data field. The user is responsible for calculating the right address and the method is highly unsafe and is strongly discouraged. Modifications last until the system is restarted (warm start).

User SYSTEM may also use the @SINTRAN-SERVICE-PROGRAM to modify symbolically named subfields (displacements) in a datafield; the *CHANGE-DATAFIELD command is used. This allows modification to the memory image (recovered at a "warm start") and the save area (recovered at a "cold start").

6.4 The queues

6.4.1 The execution queue

The execution queue contains RT descriptions of programs ready for execution when the CPU is available. All other requests for system resources have been satisfied.

The execution queue is linked through the WLINK location in the RT description, which contains the address of the RT description of the next program in the queue. The head of the queue, BEXQU, is found in location 13B in the resident part of Sintran. The WLINK location of the last element in the queue points back to BEXQU.

The queue is ordered with respect to priority. The highest priority program is first in the queue. A program is initially entered in the queue by the RT call and remains until it completes, is forcefully removed through the ABORT call or enters another queue waiting for a resource. As soon as the resource is granted, the program reenters the execution queue.

When a program enters the queue it is placed before programs with lower priority and behind programs with the same or higher priority.

A program in the execution queue may be waiting for an I/O operation to complete. It will keep its position in the queue, but the 5IOWT bit in the RT description will be set. When the monitor searches the execution queue for a program to be started, all programs with the 5IOWT bit set will be skipped.

The RT description address of the program currently executing is found in location 7 and 10B in resident Sintran. (These two are equal except during segment adminstration by the monitor.) If a higher priority program is in IOWAIT, the currently executing program is not the first in the execution queue.

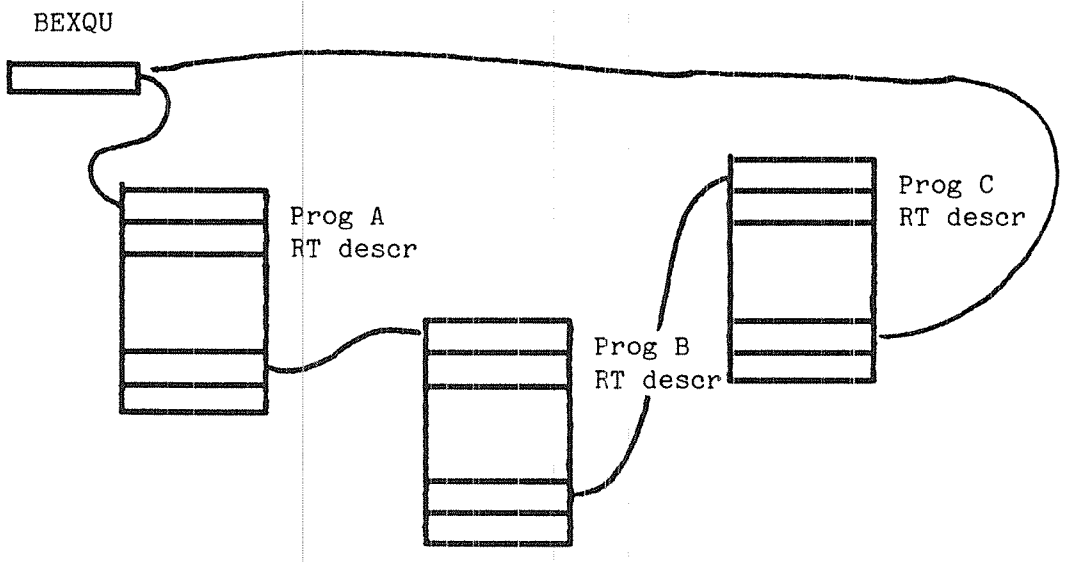


Fig. 10. Execution queue

6.4.2 **The monitor queue**

When an I/O operation is requested, the data field is entered into the monitor queue while the operation is performed and the requesting program suspended. MFUNC will point to a monitor routine restarting the RT program when the operation is complete.

The head of the monitor queue, MQUEUE, is found in location 11B in resident Sintran. This points to the datafield of the device most recently inserted in the queue. The elements are linked through the MLINK location in the datafield. The MLINK location of the datafield least recently inserted contains the value -1.

The queue is handled in a first in, first out manner. Insertion of an element is fast, removal requires that the queue is searched to the end. The monitor queue rarely contains more than one element, so the overhead is tolerable.

6.4.3 **The waiting queues**

Each device may be reserved by one RT program at a time, but several programs may have requested the device. Programs not yet granted access to the device are linked together in a waiting queue. Each device has its own waiting queue, headed by the BWLINK location in the datafield.

The queue consists of programs linked through the WLINK location in the RT description, ordered by the priority of the programs requesting the device. A program entering the queue is placed before programs in the queue with lower priority, but after those with the same or higher priority. The WLINK of the last program in the queue points back to the datafield of the device.

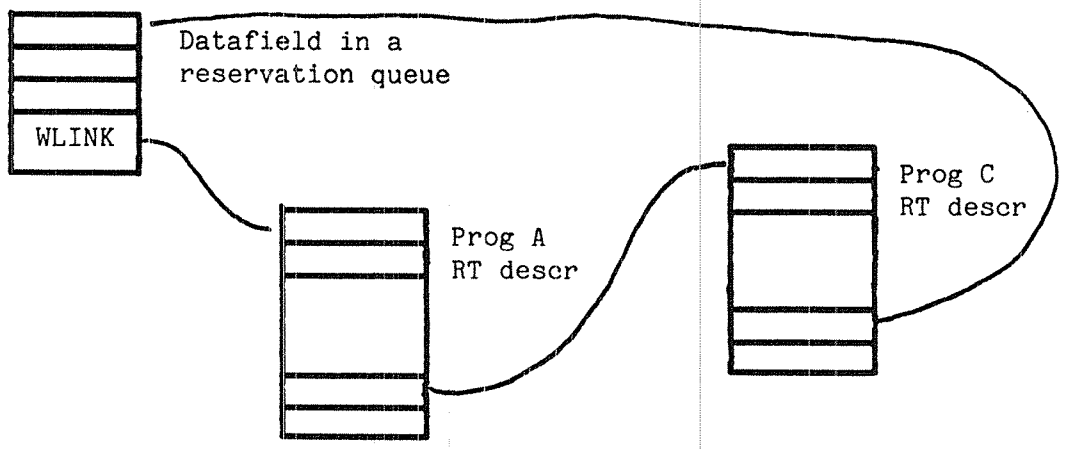


Fig. 11. Waiting queue

A high priority program does not force a lower priority program currently using the device to give up the device. The priority applies only to the order of waiting. There are monitor calls to force a program to give up a device (see section 10.6).

The RTRES location of the datafield points to the RT description of the program currently using the device. 0 indicates that the device is available.

6.4.4 The reservation queues

Each RT program may have reserved several devices. These are queued in a reservation queue, one for each RT program. The reserved devices are linked in a list headed by the BRESLINK location in the RT description and go through the RESLINK location of the datafields. The last element in the queue points back to the reserving RT program.

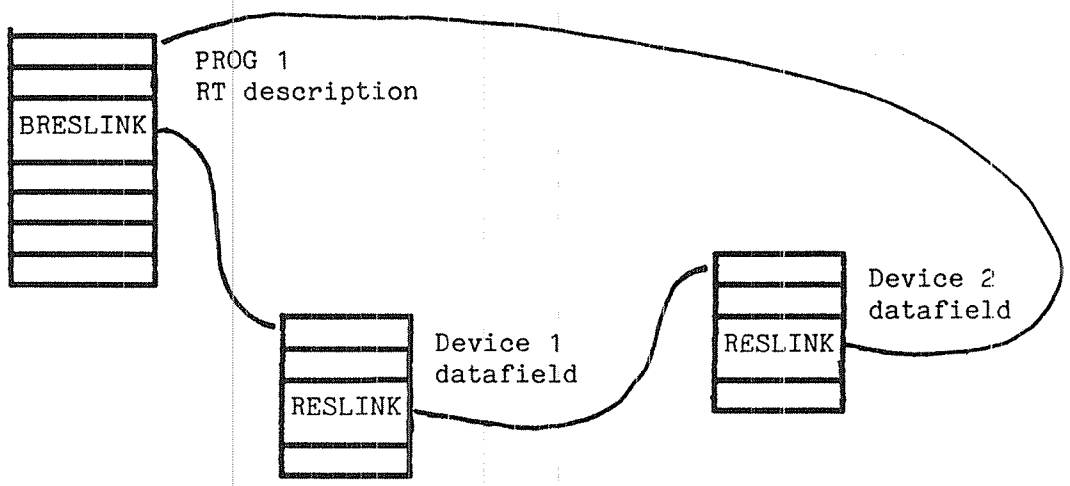


Fig. 12. Reservation queue

The reservation queue of a program can be listed by the @LIST-RT-DESCRIPTION command. Reserved devices are identified by their datafield address.

6.4.5 The time queue

A program can be scheduled to start or continue execution at some time in the future. Until this time the RT description is in the time queue.

This queue is handled similarly to a waiting queue, but rather than waiting for a resource to be available, the programs are waiting for the clock to reach a certain time when the program enters the execution queue.

The time queue is headed by BTIMQU, location 12B in resident Sintran. It is linked through the TLINK location in the RT description; the last element contains -1 in the TLINK location. The queue is ordered with respect to time - the scheduling time for each program is found in the DTIM1 and DTIM2 locations in the RT description.

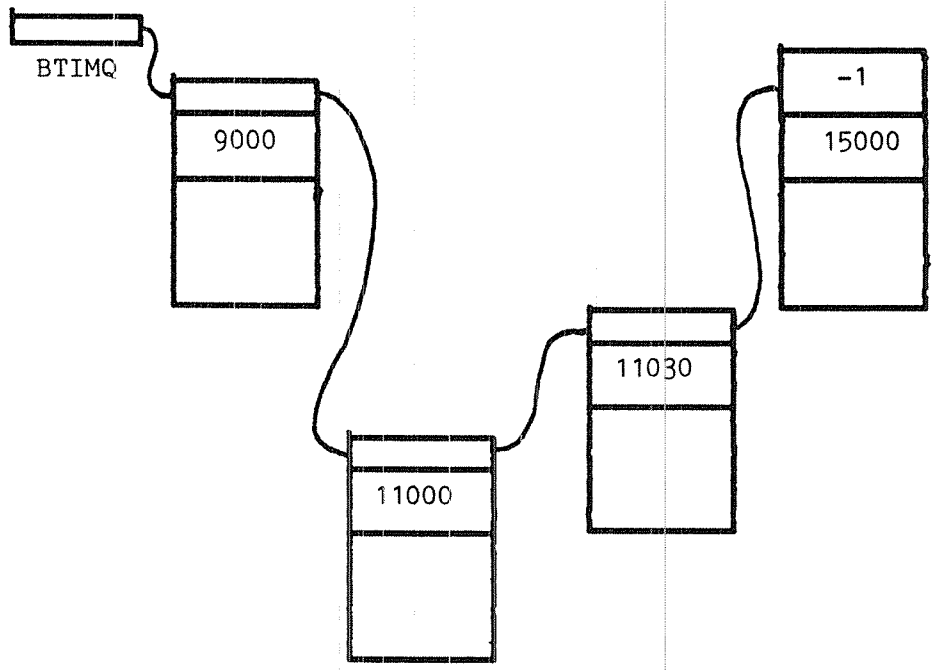


Fig. 13. Time queue

Because the time queue has its own location in the RT description, a program can be in the time queue and also in the execution queue or a waiting queue. (The execution and waiting queues are linked through the WLINK location in the RT description.) The presence of the program in the time queue does not affect execution until the waiting time has expired. If the program is active at that time it is set up for another execution as soon as it completes, otherwise it is entered in the execution queue immediately.

Commands and monitor calls can specify time relative to the current time but this is converted to absolute time, represented in internal time units, in the DTIM locations.

6.5 The operations performed on the queues

6.5.1 Initial program activation

When a program is initially activated through the RT call,

- the execution queue is scanned from the beginning until a program with a lower priority is found.
- the RT description is entered in the execution queue before this program.
- if the new program enters the queue at the front, it has the highest priority in the queue and is granted the CPU. The status of the program that was executing is saved in the RT description. The page tables, PCR register and the register block are loaded with new values for the new program and the new program is allowed to start execution
- otherwise, the program remains inactive in the execution queue until all programs ahead of it have left the queue.

6.5.2 Requesting a device

When a device is requested through the RESRV monitor call,

- the RTRES location in the datafield is inspected. If it is zero, the device is allocated immediately. Otherwise
- the chain starting at BWLINK is followed through WLINK in the RT descriptions in the waiting queue to the end,
- WLINK in the last element is set to point to the requesting RT program,
- WLINK of the requesting program is set to point to the datafield of the device.

6.5.3 Reserving a device

When a device is granted to a program,

- the BWLINK of the datafield is moved to the next element in the waiting queue of the device (if the waiting queue is empty, BWLINK is set to point to the datafield itself)
- the RTRES location is set to point to the RT program being granted the device,
- the BRESLINK location of the RT description is copied to the RESLINK of the datafield and the datafield address moved to BRESLINK
- the execution queue is scanned from the start until a program with a lower priority is found,
- the RT description is entered in the execution queue before this program.

6.5.4 Releasing a device

When an RT program releases a device through the RELES monitor call or the device is forcefully taken by the PRLS call,

- the device is unlinked from the reservation queue by setting BRESLINK or the RESLINK location of the datafield preceding the released device to the one following the released device,
- RESLINK and RTRES of the datafield is set to 0 if the waiting queue is empty, otherwise the device is granted to the first program in the waiting queue.

6.5.5 Terminating a program

A program may terminate voluntarily through the EXIT or QERMS monitor call or be forcefully terminated by the ABORT call. In either case,

- the program is searched for in the execution and waiting queues,
- it is unlinked from the queue where it is found by setting the WLINK location of the RT description preceding it to point to the description following the terminating program,
- all devices in the reservation queue linked to the BRESLINK location are released.

6.5.6 Scheduling a program for execution

A program is scheduled for execution at a specified time through the ABSET or relative to the current time through the SET call. A program may suspend itself for a certain period and automatically continue execution when the period expires, through the HOLD call. These are treated the same way,

- if the activation time is specified relatively, this is converted to absolute internal time,
- the time queue is searched for the first program in the queue with the same or later activation time,
- the program to be activated is inserted before this program, by setting the TLINK location of the program to be activated to point to the found program and the TLINK location of the program preceding the found one to the program to be activated
- the absolute time of activation is entered into the DTIM1 and DTIM2 locations of the program

6.5.7 Activating a program in the time queue

Every 20 ms an interrupt on level 13 starts which checks the time queue. If the internal time is equal to or larger than DTIM1 and DTIM2 in the first program in the queue, this program should be moved to the execution queue,

- BTIMQ is updated to point to the next program in the queue (the one pointed to by TLINK of the first program in the queue),
- if the unlinked program is already in the execution or a waiting queue, the repeat bit in the RT description is set so it is reexecuted as soon as it completes,
- otherwise it is entered in the execution queue in the same manner as when a program is started immediately
- DTIM1 and DTIM2 of the new first element of the execution queue are checked to see if this program should also be activated, in which case the same action is taken with this program. This is repeated until a program is found with an activation time greater than the current time.

6.6 Segment management

The queues used for allocating memory are completely invisible to programmers. They can be influenced as an indirect effect of other commands or with a few special commands (FIX, FIXC, UNFIX), but in most cases queue manipulation is implicit in the ordinary execution of the program. The segment manipulation routines detect situations requiring queue modification.

The page tables described in chapter 3 must be considered a part of the system tables used for segment management. This chapter describes the software tables containing the information kept in the hardware tables.

6.7 Queue elements

6.7.1 Segment table entry

Each segment in the system is described by a 5 word descriptor in the segment table. The size of the table is equal to the number of segments in the system; it is a system generation parameter with a maximum of 376B including the system included segments.

The words of a descriptor are

SEGLINK	link location to form the segment queue
BPAGLINK	head of the segment's page queue
LOGADR	logical start page of segment, segment size
MADR	start page of segment within segment file
FLAG	miscellaneous switches

LOGADR has 3 subfields:

bits 0:5 :	first virtual page number
bits 6:7 :	page table
bits 10B:16B :	size of segment in pages

MADR contains the page number within the segment file in the lower 14 bits and the segment file number in the upper 2 bits.

FLAG contains the following flags:

bit 0 :	5OK - segment has sufficient number of pages in memory to start execution
bit 1 :	5DEMAND - demand segment if set
bit 2 :	5FIX - fixed in physical memory
bit 3 :	5INHB - inhibit use (RT loader scratch variable)
bit 4 :	5SYSEGM - system segment
bit 5 :	5SPROT - protect flag
bit 6 :	5SREEP - reentrant subsystem segment
bits 7:10B :	Not used

bits 11B:17B : Protect bits (RWF), ring bits

A segment table entry is selected by the segment number in the RT description or by a call parameter.

The segment table entry can be printed in symbolic format by the SINTRAN III command @LIST-SEGMENT. This command is available to all users.

```
@LIST-SEGMENT 210

FIRST PAGE:      200 LENGTH:       33
SEG.FILE:          0 MASS. ADR:  1413
WPM RPM FPM REE-SUB DEMAND OK
```

FIRST PAGE gives the page table number (leftmost digit) and the first logical page number (two rightmost digits) on that table. LENGTH gives the segment size, and MASS. ADR the page number within the segment file. (In this system, segment 210 is a reentrant subsystem containing the BASIC compiler, all access is permitted and like all reentrant subsystems it is a demand segment.)

A numerical dump can be written by user SYSTEM through the @SINTRAN-SERVICE-PROGRAM command *DUMP-SEGMENT-TABLE-ENTRY.

```
@SINTR-SERVICE-PROGRAM

*DUMP-SEGMENT-TABLE-ENTRY
SEGMENT NO.: 210
MEMORY, IMAGE, SAVE-AREA OR SEGMENT? M
OUTPUT FILE: TERMINAL

SEGLINK:         0
BPAGELINK:       0
LOGADR:      15600
MADR:         1413
FLAG:       160103
```

The third way to list the information about the segment is the RT-LOADER command *WRITE-SEGMENT:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*WRITE-SEGMENT 210 TERMINAL

 210       0  65777  1413  0  0  2 RFW DEMAND REE-SUB
*
```

If default is used for the segment number, information about all segments is listed and a heading identifying the columns is supplied. S.NO. indicates segment number, L.ADR and U.ADR the lower and upper logical address of the segment, respectively, M.ADR is the page number within the segment file of the first page in the segment, SF is the segment file number, RI the ring and PT the page table to be used.

```
*WRITE-SEGMENT,,,
S.NO.  L.ADR   U.ADR M.ADR  SF RI PT

   2      0  73777     0   0  2  1 RFW DEMAND
   3 100000 173777    40   0  2  0 RFW DEMAND
   4 100000 173777    76   0  2  0 RFW DEMAND
   5  74000  77777    36   0  2  0 RFW NON DEMAND
   6 100000 173777   134   0  2  0 RFW DEMAND
   7 100000 123777   172   0  2  0 RFW DEMAND
  10      0  77777   204   0  2  2 RFW DEMAND
  11      0   7777   244   0  2  1 RFW DEMAND
  12      0      0                 EMPTY SEGMENT
  13 100000 147777   255   0  2  0 RFW DEMAND
  14 100000 111777   250   0  2  0 RFW DEMAND
                .
                .
 200      0 177777  5613   0  0  2 RFW DEMAND
 201  70000  77777  5713   0  2  0 RFW DEMAND
 202      0 177777  5717   0  0  2 RFW DEMAND
 203      0  21777   773   0  0  2 RFW DEMAND REE-SUB
 204      0  77777  1004   0  0  2 RFW DEMAND REE-SUB
 207  36000 177777  1307   0  0  2 RFW DEMAND REE-SUB
RTCOMMON AREA:  172000 177777
```

6.7.2 Memory Map Table entry

Each page in physical memory has a three word entry in the Memory Map Table describing the use of that page. The three words are

PAGLINK	link location for page queue
ALOGNO	logical number of page
PAGPHYS	physical number of page and flags

In Sintran VSE/VSE-500 the Memory Map Table entry size is four words; PAGPHYS is two words.

The size of the Memory Map Table is approximately equal to the size, in pages, of physical memory in the configuration.

6.8 The queues

6.8.1 The segment queue

All segments with pages in memory are linked together in the Segment queue which is ordered with respect to access. The most recently used segment is first in of the queue, the least recently used one is last.

The queue is reorganized (if appropriate) every time execution switches to a new segment because of a transfer of control by the executing program or because another program is activated.

6.8.2 The page queue

Each segment has a queue of pages starting at the BPAGLINK of the segment description. This queue contains the pages of the segment present in physical memory, linked through the PAGLINK location.

The ALOGNO gives the logical number of the page, PAGEPHYS the physical number. When the page entries are entered into the PTs, PAGPHYS is entered in the PT entry selected by ALOGNO.

Pages belonging to the segment that are not in the page queue are swapped out on disk. If the segment has no pages in memory - because no program in the segment is active or all pages have been removed from memory - the segment entry is removed from the segment queue.

The most recently used page is first in the queue (closest to the segment entry), the least recently used one is last. When a page is removed to make room for another one, the least recently used page is swapped out first. It is also removed from the page queue (and the hardware page tables if present there).

6.9 The operations performed on the queues

6.9.1 Placing a segment in memory

A segment is placed in memory if a program in the segment is started (RT call) or an explicit monitor call requests the segment (FIX, FIXC cause immediate transfer from disk; MCALL, MEXIT, REENT modify an RT description so that access to these segments will cause a page fault and initiate a transfer).

- the appropriate segment table entry is found using the segment number in the RT description as an index to the table,
- if a nondemand segment (indicated in the segment table entry), all pages of the segment are read from disk and entered in the page queue,
- the entry found is inserted as the first element in the segment queue,
- for each element in the page queue, the physical page number and flags are entered into the page table determined by the logical page number and page table index. All other page table entries are zeroed and cause a page fault interrupt if addressed.

6.9.2 Removing a segment from the page tables

Every time the contents of the page tables are updated for another segment, the status of the segment(s) presently using them must be saved. This involves

- going through the page queue(s) of the previous segment, saving the page table entry (including PGU and WIP bits) in the memory map entry.

6.9.3 Page fault handling

A page fault occurs if the page table entry is zero, indicating that the page is not in memory but must be fetched from disk. The page is therefore not present in the segment's page queue. An interrupt to level 14 starts a routine to fetch the page and update the page queue,

- the validity of the addressed page number is checked (between the first virtual page number and the sum of the first and the size, found in the segment table entry). If invalid, the requesting program is aborted,
- an available physical memory page is found,
- a disk transfer to this page is initiated from the disk address calculated as the sum of the start address of the segment and the logical page number,
- the next program in the execution queue is started while the transfer takes place.

7 PROGRAM COMPILATION AND LOADING

7.1 The source program

A program using RT facilities looks like other programs written in the same language. Almost any program can be executed as an RT program if the appropriate commands for reserving devices and/or files are given from a terminal before execution is started.

7.1.1 Operating system service requests

However the program usually knows when the program is written that RT facilities are required. Sintran III provides such facilities through a set of monitor calls, special machine instructions identifying a request to the operating system. High level languages commonly used for RT programming (Fortran, Pascal, Planc) have a set of subroutines (procedures) or functions, which will perform the monitor call at execution time. These must be accessed as external routines. Declaration of external (imported) routines is language dependent.

Assembler (MAC) or NORD-PL programmers request RT facilities by using the MON instruction. As the argument transfer mechanism varies with the monitor call, the MAC call sequence is described for all monitor calls. Users of other languages must know the call sequence if a routine call corresponding to a required function is not available. In that case, an assembler routine can be written and linked to the high level program. Details of how parameters are transferred to external routines are given in the language manuals.

Note the following points about the call sequence:

- For monitor calls available to background programs, argument transfer is the same in background and RT

- Most monitor calls available to background programs return to the second location after the MON instruction ("skip return") in a normal return, but to the first location in an error return. File system error codes are returned in the A register.

- Most monitor calls available in RT programs only return to the first location after the MON instruction. Errors cause an error message on the error device (default: console) or leave -1 in the A register. Serious errors terminate the program.

- Most monitor calls available to background programs expect arguments in one or more of the T, A, D and X registers. If the argument is compound (string, array), the address of the argument is found in a register. A result, function or status value is found in the A register.

- Most monitor calls available to RT programs only expect the A register to point to an argument list containing the addresses of the arguments.

There are exceptions to these rules; these are mentioned when the particular call is described.

7.1.2 Notation of special properties

Extensions to the standard syntax of some high level languages, particularly Fortan, allows the (initial) program priority to be specified at compile time. This is done in the Fortran PROGRAM statement:

```
PROGRAM RTTEST, 40
```

If there is no PROGRAM statement, the program name is MAIN (FTN) or #MAIN (the FORTRAN-100 ANSI-77 compiler) and the priority is one; there can be only one "unnamed" program in the system (otherwise there would be a name conflict). The priority of an unnamed program must be set by a command or another program.

7.1.3 Compile time initialization of variables

Fortran allows the specification of initial variable values through the DATA statement; most other languages have similar mechanisms.

These initial values are valid the first time the program is executed. However, when the program terminates, the current values are the initial ones in the next execution. There is no fresh backup copy of the initial program version, analogous to the :PROG file of a background program.

Therefore, if a variable should have an initial value every time the program is executed, the variable must be explicitly assigned a value (through an assignment statement) before the variable is otherwise used by the program.

7.1.4 Variable number of parameters

Some monitor calls, e.g. MAGTP, have a variable number of parameters; a function code in the argument list will indicate which parameters are significant.

The actual parameter values are fetched in the call, rather than when the values are used; the address must be valid even if the supplied value is a dummy one. In a background program, any 16 bit address is valid; an RT program may use segments not covering the entire address area. If dummy parameters are not supplied to the Fortran interface routines, random addresses that may be illegal will be supplied, causing an illegal page fault interrupt. Therefore, dummy parameters should never be omitted from the argument list.

7.2 Compilation

No special precautions are necessary when the program is compiled. All language compilers for Norsk Data computers are common for RT and background programs. In certain cases space requirements and performance, in particular I/O performance, may be affected by compiler options. Compiler options are explained in the respective language manuals.

The only exception to the main rule is if a Fortran routine uses recursive techniques or the routine should be reentrant at execution time. Then the command

```
$REENTRANT-MODE
```

is given to the Fortan compiler prior to execution. Reentrant Fortran programs do not conform to the ANSI Fortran standard, but is an Norsk Data extension. Fortran programs that do not use recursion or reentrancy may, but need not be, compiled in reentrant mode.

Programs should be compiled with the DEBUG option off. However, it is possible to load files compiled with DEBUG to a :BPUN file with the NRL background loader and read the :BPUN file to a segment. The RT loader will accept files compiled with the SEPARATE-CODE-DATA option on, but is incapable of loading code and data to separate areas (segments).

7.3 The RT-LOADER

All manipulation of segments and RT programs is done through the RT loader, rather than through the ordinary background loader (NRL). However, the same relocatable format (BRF) is used by both loaders and subroutines and programs not using RT facilities may also be loaded by NRL in order to be executed as a background program. Conversely, a program that may run as a background program may also be loaded by the RT loader and run as a real time program (but in most cases requires commands to be given to determine priority etc.).

The RT loader is available only for users RT and SYSTEM; it may only be used by one user at a time. This is to to prevent several users from updating the segment file(s) concurrently, causing inconsistencies or loss of data. If a user tries to enter the RT loader while it is in use, the error message RT LOADER ALREADY IN USE is returned. If this error occurs in a batch or mode job, the job is terminated. The RT loader cannot be reentered through the @CONTINUE command.

In order to prevent a job from terminating if the RT loader is busy, the Sintran command @SCHEDULE may be used. The RT loader is protected by a semaphore with number 503B. Including the command

```
@SCHEDULE 503
```

in the job file immediately after the @ENTER command causes the job to be held until the semaphore is released (by the user of the RT loader executing an *EXIT command). A decimal device number must be followed by a "D". This command may also be used in a MODE job. (If the @SCHEDULE command is executed in a MODE job or from the terminal, the terminal hangs until the RT loader is available and this state is not terminated by pressing the break or escape key!)

Example of a mode job:

```
@SCHEDULE 503
@RT-LOADER
NEW-SEGMENT 310 2 ND RW WP
ALLOCATE-AREA 310 20000 100000
END-LOAD
EXIT
```

7.4 The loading session

The main purpose of the RT loader is to allocate new segments and load BRF code onto these segments; it is started by the Sintran command

```
@RT-LOADER
```

7.4.1 Allocating a segment

Before loading starts, a new segment should be allocated using *NEW-SEGMENT. Default values are usually used for all parameters. If the segment number is not significant and default values are used for all parameters, the entire *NEW-SEGMENT command can be omitted. An implicit *NEW-SEGMENT is then allocated as soon as a loading command is executed and the segment number used is reported.

*NEW-SEGMENT (<segment no>) (<ring>) (<segment type>)
(<protection bits>) (<WP/NP>)

Command parameters:

<segment no> - default is the first free segment number. If a segment number is specified, it must be free.

<ring> - specifies the rights routines on the segment has to access other segments and is discussed in chapter 3. The value must be either 0 (default), 1 or 2.

<segment type> - determines whether the segment is a demand (DM) or a nondemand (ND) segment. Default is ND.

<protection bits> - limit legal operations on the segment in any combinations of read, write or fetch (instruction execution). Default is RWF, all access permitted.

<WP/NP> - may be specified as WP, setting the WIP bit of all pages immediately when the segment is placed in memory. Default is NP - the bit is not set until a store operation is performed.

If the default segment number is used (first free segment), a message is returned giving the actual segment number used. If a segment number was specified, this segment must be free, otherwise an error message is issued. The list of free segment numbers can be obtained through the *LIST-FREE-SEGMENTS command:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*LIST-FREE-SEGMENTS
OUTPUT FILE: TERMINAL

    216   217   226   227   230   231   232   233
    234   235   236   237   240   241   242   243
    244   245   246   247   250   251   252   253
    254   255   256   257   260   261   262   263
    264   265   266   267   270   271   272   273
    274   275   276   277   302   303   304   305
    306   307   310   311   312   313   314   315
    316   317   320   321   322
*NEW-SEGMENT 322,,,,,
```

The <ring>, <protection bits> and <WP/NP> bits determine the setting of the bits in the page tables when the segment is placed in memory for execution. The ring and protect bits are discussed in chapter 3.

If a DMA device writes data in memory the WIP bit is unaffected, as DMA devices do not use the memory management system. Similarly, if two or more CPUs access the same segment in multiport memory, the master CPU placing the segment in memory does not necessarily write to it setting the WIP bit. To ensure that actual changes in the segment are recorded on the segment file, regardless of which CPU/device performed the writing, all WIP bits can be forcibly set, so all pages are unconditionally written back before being removed from memory. This is done by setting the last parameter of the NEW-SEGMENT command to WP.

E.g. to allocate a new segment with all defaults, issue the command

```
*NEW-SEGMENT,,,,,,
NEW SEGMENT NO: 163
*
```

To allocate a data segment as segment number 164, with read and write permitted (but not fetch, as this is a data segment), issue the command

```
*NEW-SEGMENT 164,,,RW,,,
*
```

Two segments can be open concurrently for loading. If two *NEW-SEGMENT commands have been executed, the first one determines the segment used as default value when loading code and is termed the default load segment. The second one is the default segment used for linking.

If only one segment has been opened, this segment is the default load segment. Segments may have names of up to 7 alphanumeric characters given through the command *DEFINE-SEGMENT-NAME. The segment name can be used as parameter instead of the segment number. This applies to version I and later versions of the RT Loader only.

7.4.2 Loading code to the segment

BRF code is loaded into the new segment by one of the commands

```
*LOAD  <input file> <load-segment> <link-segment>
*NREENTRANT-LOAD  <input file> <link-segment>
*REENTRANT-LOAD  <input file> <link-segment> <stack length>
```

These commands all perform the same basic operation. All take as the first parameter the name of a BRF file; default type is :BRF.

*LOAD loads the specified file to the segment. It is used if the user wishes to specify all details about files to be loaded and does not want any automatic library loading. *LOAD is also used if code is loaded to two segments in one load session. Programs in other languages than Fortran should be loaded by this command in order to prevent unintended loading of the FTN library.

If no segment number(s) are specified, the default load and link segments are assumed. Before the *LOAD command is issued a load segment must have been allocated by the *NEW-SEGMENT command or used in a *NREENTRANT-LOAD or *REENTRANT-LOAD. The segment to which the code is loaded is the default load segement in subsequent load commands.

```
*NEW-SEGMENT 345,,,,
*LOAD MAC-PROG,,,
*END-LOAD
```

***NREENTRANT-LOAD** is used for loading Fortran routines which do not require reentrancy. Also it may be used when loading programs in other languages and RT monitor call routines are fetched from the nonreentrant Fortran library (This applies to Planc, Basic and Pascal).

When this command is used, *NEW-SEGMENT (with default values for all parameters) and loading of the nonreentrant Fortran library is automatic - FTNLIBR:BRF is loaded at *END-LOAD if *NREENTRANT-LOAD was the last load command executed. Thus, a minimum of commands are required in order to load a Fortran program.

If a current load segment exists, no implicit *NEW-SEGMENT is executed and the file is loaded to the current load segment. An explicit *NEW-SEGMENT command is necessary if non-default segment parameters should be used.

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NREENTRANT-LOAD FORTRAN-PROG,,
NEW SEGMENT NO: 345
*END-LOAD
*EXIT
@
```

***REENTRANT-LOAD** also loads the reentrant Fortran library (FTNRTLIBR:BRF) after each file is loaded if undefined references exist and allocates a stack area of the specified size. Default stack size is 1K words. The names of the nonreentrant routines in FTNRTLIB are deleted from the linking table; the routines are loaded again if another program is loaded which requires them.

*REENTRANT-LOAD is used when loading Fortran programs compiled in REENTRANT-MODE, allocating data areas on a stack at run time. This is neccessary if the program uses recursive algorithms or if several programs use the same routine concurrently. All routines should be compiled in $REENTRANT-MODE; mixing of reentrant and nonreentrant routines is illegal.

Each file loaded should contain one program only. If there are several programs in the file, they use the same stack and if they are active concurrently they destroy each other's data. If the programs are loaded from separate files, using several *REENTRANT-LOAD commands in succession, each program has a separate stack. The default size of the stack is 2000B words.

```
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$REENTRANT-MODE
$COMPILE PRO6, "PRO6:LIST", "PRO6"
3 LINES COMPILED . OCTAL SIZE=     20
CPU-TIME USED IS 0.1 SEC.
$EXIT

@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NEW-SEGMENT 270,,,,,,
*REENTRANT-LOAD PRO6
LINKING-SEGMENT NO.:
STACK LENGTH:
*END-LOAD
*
```

FTNLIBR:BRF and FTNRTLIBR:BRF are BRF library files, so only those units referenced are loaded. Units not required by programs already loaded are bypassed and not loaded.

7.4.2.1 The link segment

All three load commands have "link segment" as their last parameter. If a symbol is undefined in the current load segment, but defined in the link segment, this definition is used. The current load segment and the link segment are then used by the RT programs; consequently the two segments must not overlap in address space (see *SET-LOAD-ADDRESS below) or they must use different page tables. If the two segments are being built concurrently, they are located at the same page table, but the default initial load address of the secondly opened segment is be 100000B.

If a segment has previously been specified as a link segment in the current load session, this is the default link segment. Otherwise, if two *NEW-SEGMENT commands have been executed with no intervening *END-LOAD, the first one of these is the default load segment and the second the default link segment.

The link segment may also be a previously loaded segment, in which case all symbols in the RTFIL are available. This segment may be located at another page table than the load segment, but the user is responsible for using the altenative page table mechanism properly at execution time.

If linking is not wanted, <link segment> may be specified as 0. If no default link segment exists (the program uses only one segment), the parameter may be empty (default). Specifying 0 as the link segment is necessary only if a link segment has been previously used in the same loading session.

7.4.2.2 Setting the load address

To prevent the two segments from overlapping, the *SET-LOAD-ADDRESS command can be used immediately after the *NEW-SEGMENT command, before anything is loaded to it.

*SET-LOAD-ADDRESS (<segment no>) <load address>

Command parameters:

<segment no> - default is the first segment opened with a *NEW-SEGMENT command

<load address> - the lowest address from which code is loaded

*SET-LOAD-ADDRESS causes any code subsequently loaded to the segment specified to be located from <load address> and upwards. <segment no> may be zero, indicating RTCOMMON.

This command allows the user to force two segments that are active concurrently to occupy different parts of the virtual address space. E.g. if approximately 33000B words of code are to be loaded to segment 214 and approximately 60000B to segment 215, the loading session includes the following commands:

```
@RT-LOADER

*NEW-SEGMENT 214,,,,,
*NEW-SEGMENT 215,,,,,
*SET-LOAD-ADDRESS 215, 40000
*LOAD SUBROUTS 215,,
*LOAD MAINPROG 214 215
*END-LOAD
```

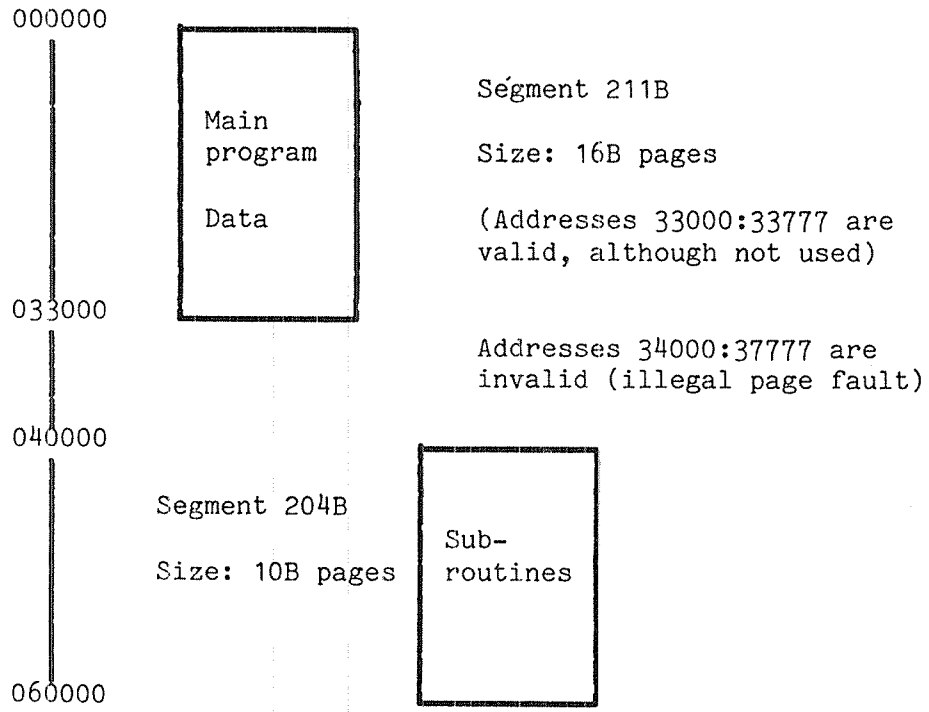


Fig. 14. Two segments used concurrently by a program

The subroutine code runs from address 40000B upwards, while the main

program is located from 0 to 33000B.

To determine the lowest legal address for the second segment, it may be necessary to trial load the first segment and see how much space is used. The current load address when all code has been loaded rounded up to the nearest page limit (multiple of 2000B) is the lowest address for the second segment.

This trial load can use the background NRL loader if the RT Loader is busy. To start loading from address 0, default load address in the RT Loader, it is necessary to use an image file, NRL command *IMAGE-FILE. The scratch file 100 may be used as image file. The lowest "FREE:.." address gives the amount of code loaded.

If two *NEW-SEGMENT commands are used during a load session, an implicit *SET-LOAD-ADDRESS is executed, setting the load address of the second segment to 100000B and an explicit command is required only if another dividing line between the two segments is desired. If the segments are loaded independently, in two sessions, the load address must be set explicitly.

*SET-LOAD-ADDRESS can be used when some code has been loaded to a segment. Code loaded after the command has been executed is located from the specified address upwards. This may leave an uninitialized area where nothing has been loaded; the initial contents of this area are unpredictable. On the other hand, the specified address may mean the new code overlaps that previously loaded. The current load addresses can be obtained through the command

```
*WRITE-LOAD-ADDRESS
SEGMENT NO.: 270

L.ADR:  20000     U.ADR:  24313   C.LADR: 24314
```

The upper, lower, and current load addresses are returned.

The segment number specified must be one of the segments currently being loaded. The lower and upper addresses of a previously loaded segment are obtained through *WRITE-SEGMENT.

7.4.2.3 Loading Fortran COMMON blocks

In a background program, common blocks are loaded to the upper part of the address space. This can be done without concern for the background segment size, as the :PROG file at execution time is copied to a segment which always covers the entire addressing range.

If the RT loader followed the same strategy, the size of segments using common blocks would be 64 pages, even for small programs. To save space in the segment file, common blocks are placed from the current load address when they are defined.

This is sometimes undesirable: read-only program code and read/write data are mixed, erroneous addressing can easily destroy program code and data areas cannot be placed on a separate segment for sharing with programs on other segments.

The common areas can be placed from a given address by the command *PRESET-COMMON-ADDRESS, given before any common blocks are loaded. All common areas are located from the specified address upwards:

```
*NEW-SEGMENT 270,,,,,,
*PRESET-COMMON-ADDRESS
SEGMENT NO.: 270
ADDRESS: 40000
*
```

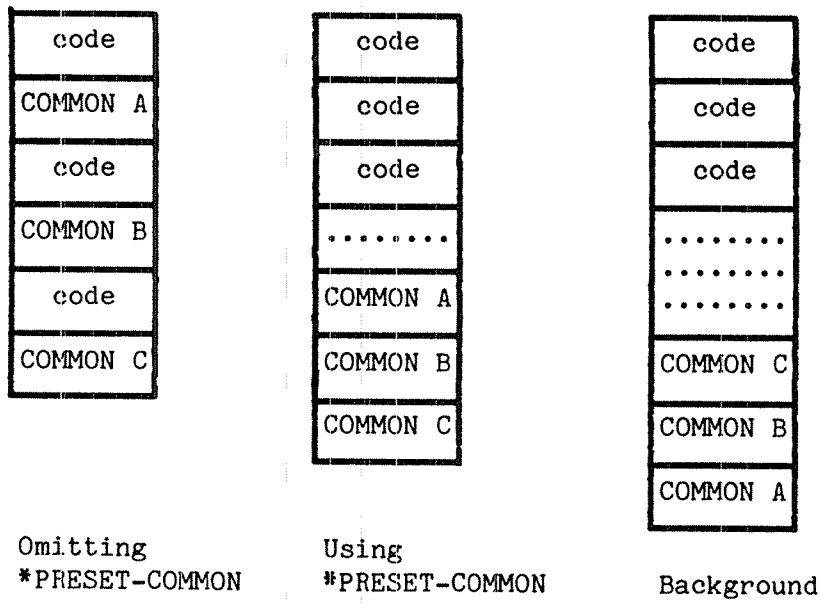


Fig. 15. Location of common blocks

To minimize the space wasted, the common address should be approximately the upper address of loaded code when all libraries have been included. If the load address when *END-LOAD is executed is much lower, there may be a lot of space between the upper code address and the lowest common address. This space is reserved, as a segment must be contiguous.

This occupies an unecessarily large area on the segment file. If not a demand segment, it also uses more memory than necessary and has a longer startup time, due to the required copying of pages from disk to memory. If an addressing error occurs during execution, it is not detected as an OUTSIDE SEGMENT BOUNDS and may go unnoticed.

COMMON blocks can also be placed on a different segment, so they are accessible from several segments. This is described in detail in chapter 14.

COMMON blocks may also be placed in the RTCOMMON area. This is described in section 12.1

7.4.3 Allocating a segment without loading to it

Situations arise when a segment should be defined without loading any code or data to it. Particularly, this applies to pure data segments where the data have no initial values, but will be written by some RT program, DMA device or another CPU (e.g. an ND-500) accessing the same multiport memory.

Also, uninitialized Fortran COMMON blocks reserve no space on the segment and other language processors may use uninitialized stack space.

A third example is when a system is using several "reentrant segments" (see chapter 15). The shadow segment is used only as a scratch area for pages that are modified and must be copied to a private segment. The initial data on this segment has no meaning and is therefore reserved without loading to it.

Before the *END-LOAD command, the required uninitialized area(s) must be allocated through the command *ALLOCATE-AREA. Unless this command is used, the upper address of the segment is the highest address to which code has been loaded when *END-LOAD is executed. At execution time, when the program attempts to access the unallocated area, an OUTSIDE SEGMENT BOUNDS error occurs.

*ALLOCATE-AREA <segment no>, <area size>, (<lower address>)

<segment no.> - the segment on which the area should be allocated.

<area size> - the size of the uninitialized area in number of 16 bit words. This parameter must be specified.

<lower address> - the start address of the area. Default is the current load address on the segment.

If there is a gap between loaded code and the specified <lower address>, this space is implicitly allocated, as non-contiguous segments are illegal. If nothing is loaded below the specified address (before or after the ALLOCATE-AREA command), the lower address on the segment is the specified one.

After the command has been executed, the current load address on the segment is the first location following the allocated area.

Example:

A program uses segment 310 as a data segment running on page table 3 for DMA transfers from disk. It should be a nondemand segment with read and write but not fetch permitted, in ring 2 for protection against less privileged programs and access to privileged devices. Because of the DMA transfer, the WIP bit should be set explicitly as soon as the segment is loaded.

The area used for DMA transfer is addressed from 100000B to 120000B and nothing else is loaded to the segment.

```
*NEW-SEGMENT 310
RING: 2
SEGMENT TYPE: ND
PROTECTION BITS: RW
WP/NP: WP
*ALLOCATE-AREA
SEGMENT NO: 310
AREA SIZE: 20000
LOWER ADDRESS: 100000
*END-LOAD
*WRITE-SEGMENT 310,,

 310 100000 117777  1062  0  2  1 RW NON DEMAND
```

7.4.4 Ending the load session

Loading to the current segment(s) must be terminated by the command

```
*END-LOAD
```

This command may not be left out; there is no implicit *END-LOAD at exit from the loader.

During load commands, the code and data is loaded to the terminal's scratch file ("file no. 100") and as new files are loaded, the code is appended at the current load address. When *END-LOAD is executed, the code is copied from the scratch file to the actual segment. Thus, the segment size need not be known until *END-LOAD; at that time the size is frozen and the segment cannot be extended.

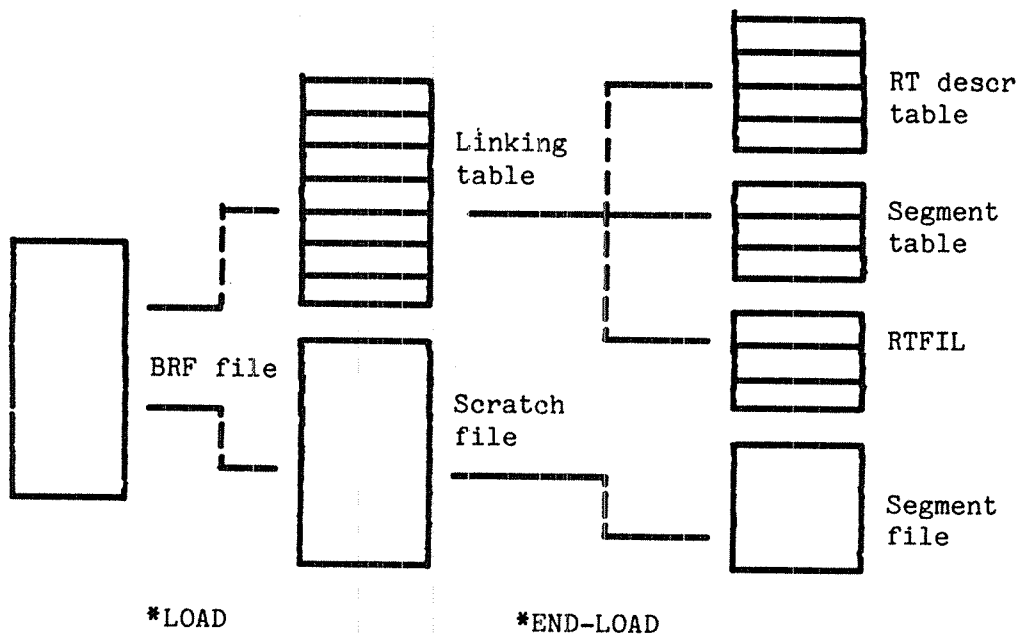


Fig. 16. Code flow during loading and *END-LOAD

The segment file is searched for a sufficiently large free area *END-LOAD. The segment table, RTFIL, the RT description table and the

segment table are updated. This, together with the copying from the scratch file to the segment takes some time - several minutes for large segments on a heavily loaded system.

If the segment file is nearly full, the user should check there is sufficiently space for the planned segment before starting to load. If sufficient space is not available at *END-LOAD, the entire load must be repeated after reorganizing the segment file or allocating another one.

*EXIT-LOADER returns control to the Sintran operating system.

7.4.5 Errors terminating the loading

Errors occuring during a mode or batch job unconditionally terminate loading. Such errors may be e.g. attempt to execute a *NEW-SEGMENT command specifying an already used segment number or fail to provide valid answers to questions.

In general, if an unexpected condition occurs the RT Loader demands that the requested operation is confirmed with a "Y". If an answer is not provided, the next command is taken as the answer, terminating the job. The "Y" must be supplied on a separate line, rather than after the parameters on the same line as the command.

To prevent the loader from terminating, the answer "Y" can be provided in the batch or MODE file. This forces the command to be executed and should be used with care! If the expected question is not asked, the "Y" is ignored. (In fact, "Y" may be treated as a "comment" command if the question is not asked and remarks may follow the "Y".)

Example of MODE job file, where any existing programs on segment 270 are unconditionally deleted:

```
@RT-LOADER
CLEAR-SEGMENT 270
Y
NEW-SEGMENT 270,,,,,
NEREENTRANT-LOAD MYPROG,,
END-LOAD
EXIT
```

If a job is terminated in error, all operations on the segments being loaded are cancelled. The segment number(s) reserved through the *NEW-SEGMENT command are not allocated and symbols defined on the segment(s) are deleted.

Operations on other segments, like defining a symbol, clearing a segment etc., executed before the error occured have already taken effect and are not cancelled. The commands to be cancelled are those executed in the *END-LOAD command, namely copying the contents of the scratch file to the segment file, updating the RTFIL table, the segment and RT description table.

Example of a job terminating because segment 310 is already in use:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*DEFINE-SYMB XXX 100 311
*NEW-SEGMENT 310 2 ND RW WP

PARAMETER NO.  1 IS ILLEGAL
a
ILL. COMMAND
 *** BATCH JOB ABORTED ***

@CC END OF MODE JOB
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*WHAT-IS XXX

 XXX         100 311     DEFINED SYMBOL
*EXIT
```

The symbol xxx is defined, but segment 310 is not modified.

7.5 Deleting a segment

When a segment is no longer used, it should be removed to make room for new ones in the segment file(s). This is done by the command

*CLEAR-SEGMENT <segment no>

<segment no> - the segment number of an existing segment

The segment must not be in use by a program when this command is executed. If the segment is fixed, it must be unfixed before the command is issued.

If there are RT programs residing on the specified segment, they are listed. Before the segment is cleared, the question DELETING THIS RT PROGRAM(S)? must be answered with YES:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*CLEAR-SEGMENT 240

RT-PROGRAMS ON SEGMENT:

 B
 A

DELETING THIS RT-PROGRAM(S)? Y
*
```

If the segment is a pure data or subroutine segment, no warning is given.

If the segment was protected by the "reentrant subsystem" flag, the user must confirm the command by answering "YES" to the question CLEARING "REENTRANT-SUBSYSTEM" SEGMENT?.

A segment may be protected from accidental clearing by setting the "protect flag" by the command

*SET-PROTECT-FLAG <segment no>

Before the segment can be cleared, the protect flag must be reset with the command

*RESET-PROTECT-FLAG <segment no>

7.6 Linking table and symbol maintainance

During loading the RT Loader keeps a record of which symbols have been defined and referenced. This is kept in the linking table. The *END-LOAD command writes the symbols to the RTFIL if they have not been removed prior to *END-LOAD. If the segment is later used as a link segment, the symbols written to the RTFIL are available to the new segment.

A defined symbol is also called a label. It has a defined value on a specific segment. All defined labels and the segments on which they are defined can be listed by using the command

*WRITE-SYMBOLS (<output file>)

<output file> - default value: TERMINAL

The symbols defined on RTFIL are available as external symbols when the segment is used as a link segment. They can be listed by

*WRITE-RTFIL (<segment no>) (<output file>)

<segment no> - only symbols on the specified segment are listed. Default value gives the symbols on all segments.

<output file> - default value: TERMINAL

Example:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*WRITE-RTFIL
SEGMENT NO: 273
OUTPUT FILE: TERMINAL

 A         40607  273    0
 8RTEN       170  273
 RESRV       165  273
 HOLD        163  273
 8LEAV       447  273
 8ENTR       237  273
 B         40641  273    0
 8OFIL       445  273
 8OVTB       466  273
 8RUTB       470  273
 8OVNO       446  273
 8ALTF       473  273
 8CLSB       471  273

 *
```

A label may be defined

- in the linking table by loading the definition of the label from a :BRF file, or
- in RTFIL by use of the command DEFINE-SYMBOL

The segment number on which a label is defined in the BRF file is loaded. The *DEFINE-SYMBOL command takes the segment number as the third parameter:

*DEFINE-SYMBOL <symbol> <value/symbol> (<segment no>)

<symbol>	up to 7 alphanumeric characters
<value/symbol>	an octal value up to 177777 or an already defined symbol
<segment no>	an existing segment, or one of the segments being built. Default is the current load segment.

All alphabetic characters in <symbol> are converted to uppercase. A maximum of 7 characters can be specified. The definition is immediately inserted into the RTFIL and stored permanently until the segment is cleared or the symbol explicitly deleted.

7.7 Taking backup of segments

There are several reasons for making backup of a segment:

- as security copy in case the original is destroyed
- as a way to unload data written to a data segment, eg. data written by a communications device
- to free segment numbers and SEGFIL space if the system using the segment will not be used for a while
- to "reload" systems generated by the background loader (NRL)
- to move the contents of one segment to another segment

7.7.1 Creating a backup

The command *BINARY-DUMP generates a :BPUN file containing the binary information on the segment. The bootstrap normally present in a :BPUN file is not generated.

The file generated can be read by the MAC assemblers or by the *READ-BINARY and *COMPARE commands.

The segment must not be in use by a program when the command is executed; all modified pages must have been written back to the segment file. Loading to the segment must be complete (*END-LOAD executed). If it was fixed in memory it must be unfixed.

*BINARY-DUMP <output file> <segment no>
(<lower addr>) (<upper addr>)

<output file> - file name or number. Default type is :BPUN

<segment no> - an existing segment. Segment number 0 indicates RTCOMMON

<lower addr> - lower boundary of area to be dumped. Default and lower permitted limit is the first address on the segment

<upper addr> - upper boundary of area to be dumped. Default and upper permitted limit is the last address on the segment

7.7.2 Recovering the backup

The *READ-BINARY command is used to read into a segment the :BPUN file generated by *BINARY-DUMP, the MAC command)BPUN or the NRL command BPUN.

*READ-BINARY <input file> (<segment no>)

<output file> - file name or number. Default type is :BPUN

<segment no> - an existing segment or a segment currently being loaded. Segment number 0 indicates RTCOMMON. Default is the current load segment

The information in the file is loaded according to the load address specified in the :BPUN file. If this area overlaps already loaded code this is overwritten without any warning. Make sure this does not occur accidentally!

If the file is read to an existing segment, the address area covered by the segment must include the entire area read from the BPUN file, otherwise an error message is issued and the reading not performed. The RT Loader requires the user to confirm that he wants to load to an existing segment:

```
*READ-BINARY SEG310 310
CHANGING EXISTING SEGMENT? YES
*READ-BINARY SEG311 311
CHANGING EXISTING SEGMENT? YES

TRYING TO EXPAND EXISTING SEGMENT
*
```

After the information has been read, the current load address is the first address above the area loaded from the BPUN file.

In many cases, the contents of a segment may be reloaded to any other segment, but the RT description of all programs using the segment must be updated to reflect the new segment number. However, if the segment is dynamically exchanged with other segments through the MCALL/MEXIT or REENT calls, it may be necessary to reload the backup to the same segment number as it was made from. Usually, this is known to the programmer, as he must make explicit references to segment number in his source program.

7.7.3 **Explicit allocation of RT descriptions**

Normally an RT description is allocated when a MAIN control byte is read in the BRF code when a program is initially loaded. If a segment containing programs is recovered by the *READ-BINARY command, the RT loader cannot identify the programs, they must be explicitly defined with the command

*DECLARE-PROGRAM <RT program> (<RT descr.address>)

<RT program> - name of program

<RT descr.address> - a valid and available RT description address. Default is the first free description

An RT description is allocated and can be referred to by the name given. The various subfields are zeroed and some of them must be set by the *CHANGE-RT-DESCRIPTION before the program can be started:

Example:

SEG270:BPUN is a copy of segment 270, containing the programs A and B. The start address, priority and segments used are given by the command @LIST-RT-DESCRIPTION: both A and B use segment 270 only and have priority 30B; the start address of A is 0, of B is 670. (This information must be known from a previous load of the segment.)

```
*NEW-SEGMENT 270,,,,,
*READ-BINARY SEG270 270
*END-LOAD
*DECLARE-PROGRAM
RT-PROGTAM: A
RT-DESCRIPTION ADDRESS:
*DECLARE-PROGRAM B,,
*CHANGE-RT-DESCRIPTION
RT NAME: A
PRIORITY: 30
SEGMENT ONE: 270
SEGMENT TWO:
START ADDRESS: 0
*CHANGE-RT-DESCRIPTION B 30 270,,670
*END-LOAD
*
```

7.7.4 Comparing backup and original

The backup created by the *BINARY-DUMP command may be compared to a segment in the SEGFIL, to detect corruption of the segment or modification of data values. This is done through the command

*COMPARE <segment no> <file>
(<lower addr>) (<upper addr>) (<output file>)

<segment no> - the possibly modified segment

<file> - a file created by *BINARY-DUMP. Default type is :BPUN

<lower addr> - lower address to be compared. Default is the lower address on the segment

<upper addr> - upper address to be compared. Default is the upper address on the segment.

<output file> - default: TERMINAL

The segment may be compared to any BPUN file, created by the RT Loader, MAC or NRL. There is little point in comparing with a file that was not initially equal to the segment, i.e. either a dump of the segment or the code initially loaded into the segment through *READ-BINARY. Each location that differs generates a one line report on the specified output file:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NEW-SEG 273,,,,
*NREENTRANT-LOAD (REAL)A,,
*END-LOAD
*BINARY-DUMP SEG-273 273,,,
*EXIT
@RT A
@RT B
```

(A and B are the programs loaded to segment 273.)

After program execution:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*COMPARE
SEGMENT NO: 273
BINARY FILE: SEG-273
LOWER ADDRESS:
UPPER ADDRESS:
OUTPUT FILE: TERMINAL

  ADR      SEGMENT     FILE

   60        237         0
   64         72         0
   65         75         0
   66         24         0
   72         34         0
   73         31         0
   74         31         0
  146        237         0
  152        160         0
  153        163         0
  154        114         0
  160        122         0
  161        121         0
  162        121         0
  445        100         0
*
```

7.8 Information about loaded programs and segments

Information about segments, RT programs and the segment file(s) can be obtained through RT Loader or Sintran commands. Some of the information can be accessed by user SYSTEM through the @SINTRAN-SERVICE-PROGRAM. The output format and the selection of information is somewhat different in the RT Loader and @SINTRAN-SERVICE-PROGRAM for most commands.

7.8.1 Segments

The lower and upper addresses, mass storage address, segment file number, ring and page protection, page table and segment type can be listed through the RT Loader command *WRITE-SEGMENT, the Sintran command @LIST-SEGMENT, or the @SINTRAN-SERVICE-PROGRAM command *DUMP-SEGMENT-TABLE-ENTRY (user SYSTEM only). All three commands are described in section 6.7.1

7.8.2 RTFIL

RTFIL symbol names, values and segment number where defined can be listed by the RT Loader command *WRITE-RTFIL. The use of this command is described in section 7.6

7.8.3 RT programs

An RT description is written by the Sintran command @LIST-RT-DESCRIPTION or the @SINTRAN-SERVICE command *DUMP-RT-DESCRIPTION. Examples of both are given in section 6.2.5

The RT Loader command *WRITE-PROGRAMS or the Sintran command @LIST-RT-PROGRAMS list all defined programs.

The addresses of unused RT descriptions are given by *LIST-FREE-RT-DESCRIPTIONS. Examples of these commands are given in section 7.4.1

7.8.4 Segment files

Segment file usage and distribution of available space can be determined by inspecting the bit file using the RT Loader command *DUMP-SEGFILE-BITMAP. This command is described in section 5.6

8 PROGRAM PRIORITY AND MEMORY ALLOCATION

The performance of a program is largely dependent on the resources time, space, external and internal devices allocated to the system. Total resources can be distributed among the requesting programs by commands or monitor calls.

CPU power and memory space allocation are invisible to the program, except for the (wall clock) time used. The operating system can manipulate priorities to a large extent. This is not usually the case with devices, as most programs request the device explicitly and need exclusive access until they voluntarily give it up.

The calls directing use of memory are

@FIX	/	MON 115
@FIXC	/	MON 160
@UNFIX	/	MON 116

In addition, the third parameter of the *NEW-SEGMENT command in the RT loader, segment type (demand/nondemand), and the system variable MAXP influences memory allocation. (MAXP limits the number of pages a segment may have in memory concurrently and is set by user SYSTEM by the @SINTRAN-SERVICE-PROGRAM command *CHANGE-VARIABLE.)

Affecting CPU priority is

@PRIOR	/	MON 110

8.1 Limitations on the programmer

RT programmers have considerable influence on the policy enforced on user RT programs, but cannot block system supervision. The available commands and monitor calls mainly affect the queues administering the programs running on level 1 and the use of page tables 1, 2 and 3.

The RT Monitor runs on level 3; it takes priority over any user RT program. See chapter 4 for a description of the interrupt system and the different program levels.

System parameters may be set to control the **maximum** memory space allocated to one program. An attempt to violate such limits causes the offending program to be terminated or recieve an error status.

The use of PT0 is partially controlled by the RT loader. Program cannot be loaded to addresses below page 40B on page table 0, as they would conflict with resident Sintran.

8.2 Memory allocation

The memory allocation commands and monitor calls affect one segment at a time; the same strategy is followed for all pages in the segment. An RT program may affect the memory resources for any segment and the last command/monitor call executed affecting allocated resources will apply.

There are three main methods of memory allocation:

- DEMAND allocation. If a page fault occurs during execution, the missing page is fetched from disk and loaded into memory.
- NONDEMAND allocation. As soon as the segment is loaded to memory, all pages are fetched and no page fault occurs.
- FIXING a segment. Demand/nondemand is a property of the segment, but fixing is the result of an explicit request to load the program to memory and have it remain until unfixed through another command monitor call.

Memory allocation is a property of the segment which applies to all programs using the segment. If a page is in memory, it is used by all programs accessing that page. If one RT program fixes a segment in memory, all users of the segment have the same advantage of speed as if they had fixed the segment themselves.

8.2.1 Demand allocation

Demand paging must be explicitly requested when the segment is loaded, as the third parameter to the *NEW-SEGMENT command in the RT loader:

```
*NEW-SEGMENT 206,,DM,,,
```

Segment number 206 is a demand segment (DM indicates DeMand).

When a required page is missing in memory a page fault is generated by hardware. The operating system finds the disk page of the missing page and starts a transfer. The RT program is suspended during the transfer and another one activated. Thus, the response time of the program largely depends on the number of page faults during execution.

A demand segment cannot be fixed in memory through a command or monitor call.

8.2.2 Nondemand allocation

Nondemand allocation is default for RT programs, but can be explicitly requested in the *NEW-SEGMENT command of the RT loader, the third parameter equal to 'ND':

```
*NEW-SEGMENT 205,,ND,,,
```

Segment number 205 is a nondemand segment. Whenever one page of the segment is used by an RT program, all its pages are copied to memory.

Startup time is significantly longer for the first program using the segment after it is written back to disk, as more pages must be read into memory. Once in memory, response times are fast and predictable, as page faults do not occur. This applies to all programs on the segment or using the segment, including those activated after the segment is in memory.

When the programs using the segment terminate or the memory is needed by other segments, some pages from the segment may be swapped out.

Nondemand segments should not be used unless required. They occupy more system resources (memory) as the number of pages available for swapping by other processes is reduced.

To fix a segment in memory, it must be a nondemand segment.

8.2.3 **Fixing**

Fixing a segment resembles nondemand allocation, but is the result of an explicit request. A nondemand segment is swapped out when no longer needed, but a fixed segment remains until unfixed. It is even more demanding on system resources and limits the freedom of the RT Monitor.

An RT program on a fixed segment has the shortest possible start up time, as no copying from disk is required. No page faults occurs and the program executes continously until another program with a higher priority enters the execution queue, it requests an I/O operation or terminates.

Before backup can be made (see chapter 7) or the segment deleted, it must be unfixed.

8.2.4 **Fixing a segment in memory**

```
Sintran command:  @FIX <segment no>
Monitor call:     MON 115  % FIX
```

The FIX call causes a segment to be copied from disk to memory and prevents it from being swapped out. The pages of the segment may be scattered in physical memory.

The @FIX command takes an octal segment number as argument.

Example:

```
@FIX 300
```

Segment number 300B is fixed in memory.

MON FIX expects the A register to point to an argument list containing the argument addresses. The only argument is the segment number, which must be a nondemand segment. If any error occurs (the segment does not exist or is a demand segment), the calling program is aborted with an error message printed on the error device.

Example:

```
        FIX=115

        LDA (PAR            % A = address of pararameter list
        MON FIX             % No error return
         ...
PAR,    (300                % Segment number 300 octal
```

In Fortran, MON 115 is available as a subroutine:

```
        CALL FIX(300B)
```

Default number base in Fortran is decimal and octal segment numbers should be followed by a B as shown above.

8.2.5 Fixing a segment in contiguous memory

The FIXC monitor call is an option which must be specified when SINTRAN III is ordered.

```
Sintran command: @FIXC <segment no> <page address>
Monitor call:    MON 160   % FIXC
```

Command parameters:

<segment no> - a nondemand segment, octal

<page address> - number of first physical memory page to which the segment is loaded

The FIXC call fixes a segment similarly to FIX, but in addition the segment is fixed in a contiguous area of physical memory, starting at a specified physical address.

An error message is issued if the segment is nonexistent or a demand segment or if any of the physical pages from the specified page up to the highest one required by the segment are already in use. Error return from the monitor call occurs only if bit 17B of the segment number parameter is also set, and leaves an error code in the range -1:-6 in the A register (see the SINTRAN III Reference Manual ND-60.128).

Contiguous fixing is used in multiprocessing applications, where several processors access the same (multiport) memory and for special I/O requirements where the external device reads or writes directly to a specified physical memory location. DMA access do not use the memory management system and if more than one page (or an area partly occupying more than one page) must be transferred, the area must be located in physically consecutive pages.

Example:

```
@FIXC 340,150
```

Segment 340B is placed in memory starting at page address 150B (physical address 320000B).

```
CALL FIXC(177B,270B)
```

The segment number 177B is fixed in memory starting at physical address 560000B.

8.2.6 Removing a fixed program from memory

Sintran command: @UNFIX <segment no>
Monitor call: MON 116 % UNFIX

Command parameter:

<segment no> - a nondemand segment, octal

A segment previously fixed through a FIX or FIXC command/monitor call can now be swapped out. It is not necessarily written back immediately, but is now treated as a regular nondemand segment.

A segment that has been fixed but not yet unfixed may not be cleared by the RT loader *CLEAR-SEGMENT command or written to a file with the *BINARY-DUMP command.

The format of the call is as for FIX:

Command:

```
@UNFIX 235
```

MAC:

```
        UNFIX=116

        LDA (PAR            % A = address of parameter list
        MON UNFIX           % No error return
         ...
PAR,    (235                % Segment number 235 octal
```

Fortran:

```
CALL UNFIX(235B)
```

If the specified segment does not exist or is not fixed, no action is taken and no warning issued.

8.2.7 The maximum area fixed

The number of pages available for fixing segments can be limited by a system parameter. This ensures the monitor has a at least a minimum number of pages available for swapping. The number of pages available for fixing may be modified by user SYSTEM through the @SINTRAN-SERVICE-PROGRAM command *CHANGE-VARIABLE FIXMAX:

```
@SINTRAN-SERVICE-PROGRAM
*CHANGE-VARIABLE FIXMAX 300
MEMORY? Y
IMAGE? N
SAVE-AREA? N

RESIDENT: 600
```

The maximum number of pages for fixing is reduced from 600 to 300 pages octal. The @SINTRAN-SERVICE-PROGRAM is not available to user RT.

8.3 CPU priority

The RT Monitor grants CPU power to that program with the highest priority in the execution queue, i.e. the first program in the queue which is not waiting for an I/O operation.

The queue is reorganized every 20 ms if necessary. This is done by the RT Monitor running at interrupt level 3, as opposed to level 1 for programs in the execution queue. The queue administration routine is initiated by the level 13 routine receiving a clock interrupt every 20 ms (see chapter 4).

Even a program with the highest possible priority (377B) cannot prevent the Monitor from performing the administrative tasks, although it may block any other level 1 program.

The initial value of a program can be set at compile time in a Fortran or Basic program, by following the PROGRAM statement with a comma and the priority.

In Fortran this looks as follows.

```
        PROGRAM RTTEST, 40
```

In MAC, the)9RT command is used. The MAIN control byte is generated where the)9RT command is encountered, not at the label following the)9BEG (if any). The second parameter to the)9RT command must be a defined symbol giving the priority; it may not be a numeric constant.

```
)9BEG
PRIOR=50
)9RT RTTEST PRIOR

%  Program code

)9END
)9EOF
)LINE
```

The programmed priority is associated with the main program. If a routine should be executed with a higher priority than the rest of the program, a PRIOR call to change the priority must be programmed. A subroutine written in MAC should not contain an RT command - in that case, the MAC routine is treated as a main program.

The programmed priority can be modified by commands or monitor calls. The priority of programs written in languages which have no syntax for specifying priority must be set by a command or call from another program.

It is very often difficult to decide what priority a RT program should have. There are no simple guide-lines in this matter, because the configuration and type of main workload differ from one system to another. Thus while testing RT programs one should not use higher priorities than 40 (50B) because then it is possible to stop the RT program from an ordinary terminal (see the section: The background timeslicing mechanism, below).

8.3.1 Waiting queue priority

The priority also determines the program's position in the waiting queue for a device requested but not immediately available. The program is entered before all programs in the queue with lower priority, but behind programs with the same or higher priority. If priority is modified while a program is in a waiting queue, the queue is reordered as soon as any modification occurs (e.g. an element added or removed, or the device released to the first program in the queue).

8.3.2 The range of priorities

Program priority can have any integer value between 0 and 255 (377B). A high value indicates a high priority. A program with priority 0 is never executed, because the system program DUMMY with priority 0 is always present in the execution queue. As programs are inserted in the queue behind programs with the same priority, other priority 0 programs will be waiting forever.

Several programs may have the same priority, in which case the first program in the queue executes until completion (or it is suspended for e.g. an I/O request or a page fault).

8.3.3 Changing the priority

Priority of an RT program does not usually change during execution. Any RT program may, however, modify its own or other program's priority, whether the affected program is executing or not. This is done through the Sintran command @PRIOR, or through MON 110.

8.3.4 PRIOR

```
Sintran command: @PRIOR <RT name> <priority>
Monitor call:    MON 110  % PRIOR
```

Command parameters:

<RT name> - the name or RT description address of an active program

<priority> - a priority in the range 0:255 (decimal)

Set the program priority of an RT program. <RT name> must be the name of an existing program or the (octal) address of an RT description. The command expects a decimal priority.

Example:

```
@PRIOR KLOKK,80
```

- the priority of KLOKK is set to 80.

The A register points to the list of argument addresses. The first argument is the address of the RT description, the second the priority. The RT loader knows the names of all RT program addresses, symbolic names can be used even in Fortran or MAC. The RT program name should be imported through a ')9EXT' (MAC) or 'EXTERNAL' (Fortran) declaration.

The A register (function value in Fortran) contains the old priority of the program.

The @PRIOR command expects a decimal number, but default number base in MAC is octal.

```
      EXTERNAL KLOKK
      INTEGER PRIOR,IP
          ...

      IP=PRIOR(KLOKK,80)
```

The priority of the RT program KLOKK is set to 80. IP receives the old priority. Equivalent MAC code:

```
      )9EXT KLOKK
      PRIOR=110
          ...
          LDA (PAR
          MON PRIOR
          STA IP
          ...
     PAR,(KLOKK
         (120                 %  120B = 80 decimal
```

If the RT program is specified as 0 (zero), or defaulted in the command, the program executing the call modifies its own priority. In case of a command, this is the background program of the terminal and has no effect because the priority of background terminals is dynamically modified by the operating system.

8.3.5 The background timeslicing mechanism

RT and background programs compete for the attention of the CPU and essentially the priority mechanisms are the same and the queue common. RT programs may have a higher or lower priority than background programs.

Background programs do not have a fixed priority during execution. An RT monitor routine adjusts the priority according to the CPU resources it uses.

Background programs do not run until completion before another program is allowed to start. Every 200 ms a routine in the RT Monitor checks whether the active program has consumed the CPU resources allocated to it. If it has, it loses its right to the CPU.

A program suspended by this mechanism is reentered in the execution queue, but with a lower priority. Initially the priority is 60B, where the program may use up to one second executing before it can be preempted by another program. If it has not completed within this time, it is reentered in the queue with priority 50B.

The program is entered behind all other priority 50B programs, thus if several programs run on the same priority, they are executed in round robin fashion.

Running on priority 50B the program may be active for up to four seconds before suspension by another priority 50B program, but may at any time be interrupted by a higher priority program. This gives fast response to interactive users running small jobs (e.g. executing a Sintran command), while larger resource demanding programs have to wait.

As long as the program communicates with the terminal and no other device, it remains at priority 50B. If it uses its entire time slice without terminal I/O, its priority is reduced to 40B for the next time slice.

If interaction with a device other than the terminal is initiated, the priority drops to 20B.

If a program requests the use of a resource already reserved by another program with priority 20B, the program reserving the resource is temporarily raised to priority 37B and executes ahead of all other priority 20B programs, freeing the resource more quickly.

9 ACTIVATING AND DEACTIVATING RT PROGRAMS

RT programs can be started and stopped by several commands and monitor calls. Programs can also schedule themselves for reexecution and they can stop themselves or be terminated by other programs or operators.

9.1 Starting a program immediately

```
Sintran command: @RT <RT name>
Monitor call:    MON 100  % RT
```

Command parameter:

<RT name> - name or RT description address of existing program.

This enters the program in the execution queue immediately. The program name must be an already loaded program or the octal address of the RT description of a program.

The same function can be performed by another program through the monitor call RT (MON 100). The single parameter is the address of the RT description, the A register pointing to the argument list:

```
        )9EXT PROGNAM
        RT=100

        LDA (PAR            % A = address of parameter list
        MON RT              % MON 100, no error return.

PAR,    (PROGNAM
```

Fortran:

```
        EXTERNAL PROGNAM

        CALL RT(PROGNAM)
```

The RT loader recognizes symbolic program names, declared as external symbols. These programs must be loaded before the program using their names. A program may refer to itself by giving the parameter value zero.

If the program is already in the execution queue (which is always the case if a program refers to itself), it is set up for another execution immediately after it terminates, by setting the repeat bit in the RT description.

9.2 Scheduling a program for execution

A program can be put into the time queue for execution later. The start time can be specified relative to the current time or as an absolute wall clock time:

Start execution after specified delay:

```
@SET
MON 101  % SET, delay specified as time unit, no of units
MON 126  % DSET, delay specified in basic time units
```

Start execution at specified absolute time:

```
@ABSET
MON 102  % ABSET, execution start specified by wall clock time
MON 127  % DABST, execution start specified in internal time
```

If the clock is adjusted, programs are treated differently depending on which command was used to enter them in the time queue. If entered through ABSET, execution is started according to the new time, if entered by SET, DSET or DABST, the delay before start is independent of the clock.

9.2.1 Starting execution after a specified delay

```
Sintran command: @SET <RT name> <no of units> <time unit>
Monitor calls:   MON 101  % SET
                 MON 126  % DSET
```

Command parameters:

<RT name> - name or RT description address of existing program.

<no of units> - the number of time units delay

<time unit> - a time unit size in the range 1:4 (see below)

The specified program is entered in the time queue with a waiting time as specified. <time unit> is

1 - basic time units (20 ms)
2 - seconds
3 - minutes
4 - hours

<no of units> indicates how many basic time units, seconds, minutes or hours the program should wait. The maximum number of units is 32767 (16 bits signed value).

The start time is stored as a number of basic time units since system restart. Adjustment of the clock by CLADJ has no effect on waiting time. If the program is active when the start time arrives, the repeat bit in the RT description is set, indicating that it should be reexecuted as soon as it terminates.

A program can only be entered in the time queue once at a time. If it was already in the queue when SET or DSET was executed, it is removed and reentered according to the new specifications.

If the specified number of time units is zero or negative, the program is entered in the execution queue at the first basic time unit increment (or if active, the repeat bit set).

Example:

A program may request that it be aborted if not completed within five minutes by scheduling a "murder program" for execution at that time

```
        PROGRAM P1,40
        EXTERNAL MAINP
        ABORT (MAINP)
        END

        PROGRAM MAINP, 32

C   Observe that P1 must have higher priority than MAINP
C   in order to abort it if MAINP is active

        PARAMETER (MIN = 3)
        EXTERNAL P1

        SET(P1, MIN, 5)

C   Perform whatever action MAIN is intended to perform

        END
```

In a MAC call, the A register points to the parameter list. The same two programs in MAC would look as follows:

```
        LEAVE=0
        ABORT=105
        SET=101
        MIN=3

        )9BEG
        )9ENT P1
        )9EXT MAINP
        LDA  (PAR
        MON ABORT
        MON LEAVE
PAR,    (MAINP
        )9END

        )9BEG
        )9ENT MAINP
        )9EXT P1
        LDA (SETP
        MON SET

        %  Perform desired actions

        MON LEAVE
SETP,   (P1
        (MIN
        (5
```

The only difference between SET and DSET is the specification of the time: DSET expects a number of basic time units in the argument list, rather than a time unit size and a number of units. The number of time units is specified as a double (32 bit) word:

```
        DSET=126

        LDA (SETP
        MON DSET

SETP,   (P1
        (0          % Most significant part = 0
        (35230      % 35230B = 15000 decimal = 5 min
```

9.2.2 Starting execution at specified wall clock time

If a program should be executed at a specified time of day, it may be inconvenient to calculate the delay from the current time and convert it to a number of time units. Besides, it may be desirable to tie a program to the clock time, making it sensitive to adjustments of the clock.

```
Sintran command: @ABSET <RT name> <second> <minute> <hour>
Monitor call:    MON 102  % ABSET
```

Command parameters:

<RT name> - name or RT description address of existing program.

<second> - decimal value 0:59, default 0

<minute> - decimal value 0:59, default 0

<hour> - decimal value 0:23, default 0

Time is specified according to a 24 hour clock. If the specified time has already passed when the call is executed, the program is scheduled for the next day.

If the program is already in the time queue, it is removed and reinserted at the specified time.

If the clock is adjusted by the CLADJ call and the program has not yet started, it is scheduled for execution according to the new time. If the clock was adjusted forwards, passing the execution start time, the program is not started at all. When the RT Monitor inspects the time queue it will consider the program already started, even though it has never actually been activated.

Example:

To activate the REQUEST program in the example in chapter 1 at 8 o'clock, the following command can be given at any time after 0800 the previous day:

```
@ABSET REQUEST,,,8
```

9.2.3 **Starting execution at a specified internal time**

To specify the time more accurately when starting a program after a long delay, a double integer (32 bits) value containg the internal time in number of basic time units can be used. It is sometimes necessary to schedule a program at an absolute time more than 24 hours in advance. This is done through the DABST monitor call; DABST is not available as a command.

For the higher precision to have any practical value, the priority of the program should be high enough to ensure that it is started when the time arrives without unnecessary delay from other program.

The internal time is the number of basic time units since the system was last restarted. It is not usually of interest; starting time should be specified relative to the current time. The TIME call (MON 11 - see section 9.10) returns the current internal time and the delay can be added to this value.

For starting WW3 in exactly 96 hours (17280000 units of 20 ms) another program may execute the following:

```
EXTERNAL WW3

CALL DABST(WW3, TIME(DUMMY) + 17280000)
```

9.3 Starting due to an external interrupt

Most RT programs communicate with the "outside world" through standard devices, accessed through I/O monitor calls. If the device inputs data before the program is ready to read it, the data is buffered by a Sintran "driver" and transferred to the program as soon as requested.

In most cases, the device may interrupt several times before any data is read by the program and each interrupt causes a new data value to be buffered. E.g. the normal size of a terminal input buffer allows an operator to enter up 64 characters before the program reads the first character, without loosing data. The size of the buffer is device dependent and can be reconfigured according to need.

Non-standard devices can be handled by a user written driver routine and a user written program can be activated as the result of the interrupt. This is done by setting up a connection between a program and an external interrupt source through the CONCT call.

9.3.1 Setting up the connection to the device

```
Sintran command: @CONCT <RT name> <log. unit>
Monitor call:    MON 106  % CONCT
```

Command parameters:

<RT name> - name or RT description address of the program to be started when an interrupt occurs

<log. unit> - decimal logical device number of the interrupting device

MON CONCT expects the same two parameters, the A register pointing to the parameter list:

```
        CONCT=106

        LDA (PAR
        MON CONCT

PAR,    (HANDL    % Program to be started
        (453      % Interrupting device
```

The logical unit numbers of devices are given in appendix C of SINTRAN III Reference Manual ND-60.128.

One program can be connected to several input sources and will be started when an interrupt occurs from any source. Each device must be connected to the program via a CONCT call.

The interrupt itself must be taken care of by a driver routine, running on the interrupt level to which the device is connected.

The program is activated as if an RT call was executed. If the program is in an RTWT state, execution is continued in the location following the RTWT call (see section 9.8). If the program is terminated, it is started in its initial start address, if active, the repeat bit in the RT description is set.

The interrupt handling routine is usually written in MAC, and entered on PT0 immediately following resident Sintran.

9.3.2 **Breaking the connection with the device**

A program is disconnected from the interrupt sources defined through one or more CONCT calls through the DSCNT call:

```
Sintran command: @DSCNT <RT name>
Monitor call:    MON 107  % DSCNT
```

Command parameter:

<RT name> - name or RT description address of the program that has been handling interrupts

Similarly, the only argument to the monitor call is the RT program name:

```
        DSCNT=107

        LDA (PRNAM
        MON DSCNT

PRNAM,  (HANDL
```

The call applies to all devices connected. Future execution of the program is prevented. If the program is in the time queue it is removed. If the program is active, execution is continued normally and the DSCNT call does not imply the release of any resources.

A program may disconnect itself from its interrupt sources or a terminal operator or any other program may perform the DSCNT call. A program may refer to itself by using an RT description address of zero.

9.4 Periodic execution of a program

Programs monitoring physical processes are often executed repeatedly at regular intervals. Each execution reads, and possibly analyzes, input from external sensors or writes output to contollers.

Other periodic programs may be timeout routines executed only if an expected event does not occur before the time interval has elapsed.

Periodic execution of a program is specified by

```
Sintran command: @INTV <RT name> <no of units> <time unit>
Monitor calls:   MON 103  % INTV
                 MON 130  % DINTV
```

Command parameters:

<RT name> - name or RT description address of existing program.

<no of units> - the number of time units delay (0:32766)

<time unit> - a time unit size in the range 1:4

<no of units> and <time unit> is specified as for SET, <time unit> equal to 1 indicates basic time units, 2 is seconds, 3 minutes and 4 hours.

MON INTV expects the same parameters as the command, while MON DINTV expects a double word (32 bits) defining the number of basic time units between execution starts. DINTV is used where a high precision in time is combined with a long interval - if the interval is less than approximately 10 minutes (32766 basic time units), DINTV is equivalent to INTVT.

The DINTV or INTV call must be specified before the program is started, but does not activate the program, it merely defines the interval between each execution. The first activation must be by some other means, such as an RT, SET or ABSET call.

As soon as the program is first activated, it is put both the execution and time queues. When the interval time has expired, it is put into the execution queue and reinserted in the time queue.

The interval time is the minimum time between two execution starts. If one execution is delayed, the delay is carried over to the following executions. Such delay may occur for example if the previous execution was not complete before the interval time was up.

An example is found chapter one, setting up PROMPT to be executed every 10 seconds and then activating PROMPT. This code is found in the program REQUEST (Fortran):

```
      EXTERNAL PROMPT
      PARAMETER (SEC=2)

      INTV(PROMPT,10,SEC)
      RT(PROMPT)
```

Coded in MAC, PROMPT could be started for periodic execution by the following code (all BRF directives required to make this a complete program are included):

```
)9BEG
PRIO=40
)9EXT PROMPT
)9RT PSTART PRIO

        INTV=103
        RT=100
        LEAVE=0

        LDA (PAR
        MON INTVT           % Set period to 10 sec
        LDA (PAR            % Only 1 argument used
        MON RT              % Initiate execution
        MON LEAVE           % Terminate

PAR,    (PROMPT
        (12
        (2
)9END
)9EOF
)LINE
```

9.5 **Terminating a program**

The executing program may voluntarily stop in several different ways:

```
MON 0     % LEAVE, normal termination
MON 134   % RTEXT, controlled error termination
MON 65    % QERMS, file system error termination
```

The effect of the LEAVE and RTEXT calls is the same in RT programs. The program is removed from the execution queue, all devices reserved by it are released and the repeat bit in the RT description is checked to see whether the program should be restarted. (If it should, the repeat bit is immediately reset and the program put into the execution queue again; reserved devices are released.)

The choice between LEAVE and RTEXT can be made to obtain the desired effect if the same program or subroutine is executed in background. RTEXT in a background batch or mode job causes the entire job to terminate, while LEAVE terminates only the executing program and continues with the next command in the file.

Normal use of these calls is LEAVE for successful completion and RTEXT for unsuccessful completion (controlled error termination). The Fortran compiler generates MON LEAVE at the END statement of a program; RTEXT must be explicitly programmed as a call to the routine RTEXT:

```
      IF (INDEX.GT.MAXLEGAL) CALL RTEXT

C  Termination here if indexing error
C  Rest of program or routine (not executed if RTEXT called):

      ARR(INDEX) = SOMEVALUE
       .
       .
```

RTEXT and LEAVE both allow the program to be executed again if it is periodic. ABORT, discussed in the next section, is used for serious or forced error termination and prevents repeated execution.

MON QERMS is equivalent to

```
MON ERMSG
MON LEAVE
```

MON ERMSG is discussed in detail in chapter 16. Its purpose is to print a file system error message on the error device. The error code is usually returned from a monitor call in the A register and MON ERMSG expects the error code in the A register. MON ERMSG or MON QERMS can be used as error return immediately following a monitor call - QERMS terminates the program immediately.

QERMS does not prevent repetitive execution of a periodic program (and will not terminate the job in background).

9.6 Forced program termination

RT programs occationally go into deadlock - unresolvable access conflicts causing one or several programs to "hang", monopolizing devices or other system resources.

Such programs should be forcibly stopped by an operator or another program. This is called to abort the program.

If a repetitive program is aborted, future reexecutions are inhibited. It does not resume periodic execution until an RT, SET or ABSET call is executed to start it. When restarted, the interval is not changed.

A program is aborted through:

```
Sintran command: @ABORT <RT name>
Monitor call:    MON 105  % ABORT
```

Command parameter:

<RT name> - name or RT description address of existing program. Default value is illegal.

MON ABORT expects the A register to contain the address of a location where the address of the RT description of the program to be aborted is found. A program may abort itself by specifying zero as the RT description address.

The program specified is terminated and removed from any queue that it may be in. All its reserved resources are released. The repeat bit in the RT description is reset, so that the program must be explicitly restarted.

ABORT should not be considered a normal error termination, as no error report is provided by the system. (The normal error termination is to issue an error message through ERMON before terminating by RTEXT. See chapter 16 for a description of error handling). The major use for ABORT rather than RTEXT is to prevent repeated executions.

The Fortran call is used as a standard subroutine:

```
      EXTERNAL PARTNER

C  Serious error condition has occurred - abort cooperating
C  program PARTNER first:

      CALL ABORT(PARTNER)

C  Then terminate this program and prevent repeated execution:

      CALL ABORT(0)
```

There are other ways to prevent a program from execution:

If the priority of a program is zero, it is never executed, but its reserved resources are not released. The PRIOR command/monitor call is described in section 8.3

The RTOFF call prevents the program from being restarted, but if active, it may continue execution. RTOFF is often used with ABORT.

9.7 Prohibiting program execution

Even though a program has been stopped, it may be restarted by any of the mechanisms described in the first section of this chapter. This may be undesirable for several reasons:

- the program may occupy resources that should be available for another program
- starting the program may cause a known deadlock condition
- it may be known beforehand that the program cannot complete successfully due to unavailable resources.

If the program is in the "RTOFF state", it is not started by a command, monitor call, arrival of scheduled start time or external interrupt. RTOFF state is indicated by setting the RTOFF bit in the RT description (bit 16B in the ACTPRI word). The program is set in this state through

```
Sintran command: @RTOFF <RT name>
Monitor call:    MON 137  % RTOFF
```

Command parameter:

<RT name> - name or RT description address of existing program. Default value: no action.

The standard parameter transfer mechanism applies to MON RTOFF. The A register points to the argument list, which contains the program address. 0 indicates the calling program and return is to the first location following the call (no error return):

```
        RTOFF=137

        LDA (ARG            % A = addr of parameter list
        MON RTOFF           % MON 137
        MON LEAVE           % Stop after having prevented
                            % reexecution

ARG,    (0                  % Addr = 0, this program
```

If the program is not periodic, but has been scheduled for later execution, the RTOFF bit in the RT description is checked at the time when the program would normally have been started. If the program is in the RTOFF state, it is not started and it is removed from the time queue. An explicit call is required to start it later.

The program may be active when RTOFF is executed and it continues to execute until completion. If another attempt is made to start the program while it is active, the repeat bit is set even if the program is in RTOFF state, but no reexecution of the program is performed after it has completed.

Every time the interval of a periodic program expires, an attempt is made to start the program. As long as the program is in the RTOFF state, these attempts are unsuccessful. If a program has a short interval and is inhibited for a long period, aborting the program reduces system overhead.

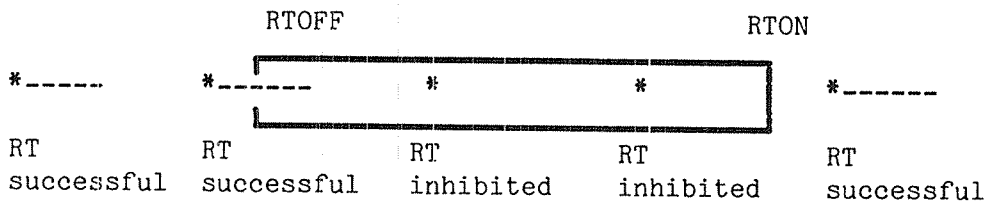


Fig. 17. MON RT inhibited in the RTOFF state

If at a later time the program should be allowed to start, the program is removed from the RTOFF state by the RTON call. This resets the RTOFF bit, permitting the program to be started:

Sintran command: @RTON <RT name>
Monitor call: MON 136 % RTON

Command parameter:

<RT name> - name or RT description address of existing program. Default value: no action.

The program can now be started by an RT call, or if the program is in the time queue, it is allowed to start when the waiting time has expired.

A periodic program resumes periodic execution, and no explicit starting of the program is required. The intervals are in step with the intervals at the time the RTOFF state was entered.

9.8 Suspending program execution

A program may execute a request to halt execution, but not terminate the program. Halt implies that

- no resources are released
- when restarted, execution continues at the instruction requesting the halt rather than at the initial start address
- the program may request restart after a specified delay

One command and three monitor calls are available for suspending execution:

```
Sintran command: @HOLD <no. of units> <unit>
Monitor calls:   MON 104  % HOLD
                 MON 267  % TMOUT
                 MON 135  % RTWT
```

Command parameters:

<no.of units> - the number of time units delay as a positive decimal integer

<unit> - a time unit size, specified as a digit in the range 1:4

<no. of units> and <unit> follow the same syntax as SET and INTV; a <unit> of 1 indicates basic time units, 2 is seconds, 3 minutes and 4 is hours.

The HOLD and TMOUT calls specify a timeout, after which the program automatically continues. The call is permitted even for background programs. The program is entered in the time queue, with a starting time determined by the arguments to HOLD and the current time. Otherwise, its effect is the same as that of RTWT and the RTWT bit of the program is set.

TMOUT will on restart indicate the restart reason in the register:

```
A =  0 : time elapsed
A =  1 : an interrupt occured, causing program to restart
A = -1 : the repeat bit (5REP) was set, causing immediate
         restart, i.e. the call caused no delay.
```

An RTWT call will not enter the program in the time queue, thus, it acts as a HOLD with an infinite timeout. A program executing an RTWT call is usually restarted by another program or a command. Periodic programs that should perform full initialization only the first time executed may use MON RTWT rather than MON LEAVE at the end of each execution. (The RTWT call should be followed by a jump to the restart address, executed when the program is restarted.)

The RTWT bit is bit 15B in the STATUS word (word 1) of the RT description. If an RT call is executed for the program, it is restarted (even if the HOLD time has not expired).

A program in a HOLD/RTWT state is removed from the execution queue until it is restarted (either by the timeout or by an RT call). If the repeat bit in the RT description is set when the HOLD or RTWT is executed, the bit is reset and execution immediately resumed.

OBSERVE:

> A program in the RTWT state will NOT release any of its reserved resources. This is of particular importance if the program is inhibited by an RTOFF; the program may continue to execute and may execute a HOLD or RTWT. It holds its reserved devices occupied and has no way to free them until an RTON call has been executed and then an RT call to restart the program (the RT call is not required if the HOLD time has not expired).

A program may have requested an I/O transfer, specifying immediate resumption of execution rather than waiting for the transfer to complete. This is called "NOWAIT mode" and is discussed in chapter 13. If the program executes a HOLD or RTWT while a transfer is going on, completion of the transfer interrupts the RTWT state and restarts the program. A @HOLD command executed in background can be interrupted by pressing the 'escape' key, or by an RT command (given from another terminal) restarting the background RT program (BAKnn for terminal, BCHnn for a batch processor).

9.9 Resetting the repeat bit

Termination of an activity executing in parallel with a program usually has the same effect as an RT call on the program, to bring it out of a HOLD or RTWT state. Example of activities restarting a program is the completion of an IO transfer in NOWAIT mode, arrival of an XMSG message or a break condition on an ND-net channel.

If the program is not in an RTWT state when the restart is attempted, the repeat bit (5REP) in the RT description is set. As soon as the program terminates it is reexecuted. In order to clear the repeat bit to prevent reexecution the program may execute a HOLD with a zero waiting time:

```
CALL HOLD(0,0)
```

This call must be executed after the repeat bit has been set, after termination of all activities that may unintentionally set the repeat bit.

9.10 Reading the clock and clock adjustments

The ND-100 counts the clock time, equivalent to ordinary wall clock time and date and internal time, which is a count of 20 ms intervals since the system was started (warm start). The offset between wall clock time (transformed to basic time units) and internal time is constant as long as the machine is running, but another offset is determined after a restart.

Most ND-100s are delivered with a panel clock which runns independently of CPU operation. When the CPU stops for any reason, this clock continues to run, and as soon as the CPU is restarted, the panel clock is read to determine the new offset between internal and clock time. NORD-10s and machines delivered without a panel clock must have the clock time updated by an operator after a stop. If there is no panel clock automatic updating is not performed, but the monitor calls to read the clock time read the value of a "software clock" whose time is derived from the internal time.

The panel clock time, usually simply called clock time, can be adjusted by commands or monitor calls; internal time cannot.

Points in time can be specified according to the panel clock or according to internal time. Adjustments in clock time may have different effects on programs, depending on whether time was specified according to internal or clock time.

9.10.1 Reading internal time

The internal time may be read using the monitor call

```
MON 11    % TIME
```

The number of 20 ms intervals since the last system restart is returned in the AD register as a 32 bit integer. In Fortran, a DOUBLE INTEGER function TIME may be used:

```
DOUBLE INTEGER TIME, TI

TI = TIME(DUMMY)
```

The argument is not used, but required by Fortran syntax. TIME is available from background as well as RT programs.

The internal time is used when starting another program by the DABST monitor call. It is also the most precise measure of elapsed time.

When a program is scheduled for execution by the SET command, the starting time is immediately transformed into internal time and the program is transferred to the execution queue when the internal time counter reaches the calculated value; clock adjustments does not affect program execution.

The internal time cannot be modified by software. However, if the system halts temporaryly due to a power failure, the time counter is not incremented during the stop period. If the system is stopped through the Sintran command

```
@STOP-SYSTEM
```

(permitted for user SYSTEM only), a power failure is simulated and the counting of clock pulses ceases until the system is restarted through the microprogram command 20! (see the System Supervisor Manual). After such a stop, the clock time is read into the CPU registers for updating the offset between clock time and internal time.

A program whose scheduling is specified in internal time units is delayed for as long as the stop lasted. This includes periodic programs; the interval between two executions is correspondingly extended.

However, after the clock has been read, the time queue is inspected for programs that should have been started during the stop period. These programs are immediately transferred to the execution queue, exactly as after a CLADJ call (see below).

9.10.2 **Reading the clock time**

The clock may be read by the monitor call

```
MON 113   % CLOCK
```

The result is returned in a 7 element array whose address is found in the location pointed to by the A register. The call sequence in MAC is

```
          CLOCK=113

          LDA (ARRAD
          MON CLOCK

ARRAD,    CLDAT

CLDAT,    0         % basic time units
          0         % seconds
          0         % minutes
          0         % hour
          0         % date
          0         % month
          0         % year
```

The call is permitted from background programs. In Fortran, the routine CLOCK can be used. CLOCK expects a seven element integer array as an argument:

```
          INTEGER CLDAT(7)

          CALL CLOCK(CLDAT)
```

When used in a program, all values are returned as integers. To display the current date and time there is a Sintran command:

```
@DATCL
18.11.14   21 SEPTEMBER 1981
```

9.10.3 Adjusting the clock

The panel clock may need adjusting for several reasons:

- the machine does not have an automatic update feature
- the machine is initially installed
- the clock is unstable due to varying temperature conditions
- a power failure lasted too long for recovery to be possible (approx 30 minutes).
- the clock should be adjusted one hour for "summer time" or "winter time"

The clock may be set by a command or a program. Adjustments may be absolute or relative to the current value of the clock.

9.10.3.1 Relative adjustment

Relative adjustment of the clock is performed by

```
Sintran command: @CLADJ <no. of units> <time unit>
Monitor call:    MON 112  % CLADJ
```

Command parameters:

<no.of units> - the number of time units clock adjustment as a positive or negative decimal integer

<time unit> - a time unit size, specified as a digit between 1 and 4

The clock is never actually set backwards by this command. If a negative adjustment is specified, the clock stands still for the specified period.

If a positive adjustment is specified, the adjustment is added to the current time. The time queue is then inspected and any program scheduled for execution at a wall clock time (through ABSET) between the old value of the clock and the new value, is immediately transferred to the execution queue.

Programs scheduled for execution by the ABSET call are affected by an adjustment of the clock.

Adjustment of the clock does not modify internal time, but it does modify the offset between internal and external time.

Programs scheduled for execution by the DABST or SET call are not affected by an adjustment of the clock.

Example:

```
@CLADJ
NO. OF UNITS: 4
TIME UNIT: 3
@CLADJ
NO. OF UNITS: 35
TIME UNIT: 2
```

This advances the clock by 4 minutes and 35 seconds. If executed from a Fortran program, the two subroutine calls are

```
      PARAMETER (SEC=2, MIN=3)

      CALL CLADJ(4, MIN)
      CALL CLADJ(3, SEC)
```

In MAC, the A register should point to the argument list:

```
      SEC=2; MINU=3
      CLADJ=112

      LDA (MINAD
      MON CLADJ
      LDA (SECAD
      MON CLADJ

MINAD, (MINU; (4
SECAD, (SEC;  (35
```

9.10.3.2 **Absolute adjustment**

The clock may be set to any wall clock time (later than January 1st, 1981) by the UPDAT call. The operator or program must specify the elements from minutes to year in the array read by CLOCK - the seconds and 20 ms counters are both zeroed.

```
Sintran command: @UPDAT <minute> <hour> <day> <month> <year>
Monitor call:    MON 111  % UPDAT
```

Command parameters:

<minute> - the new value of the minute counter, range 0:59

<hour> - the new value of the hour counter, range 0:23

<date> - the new value of the date, range 1:31

<month> - the new value of the month, range 1:12

<year> - the new value of the year, range 1950:2013

The new values are checked to confirm that they are reasonable; if they are not, the call is not executed. Examples of unreasonable arguments are negative values, second or minute values exceeding 59, or date equal to 31 in a month with only 30 days. Both command and monitor call require the year to be specified in full with four digits.

The MAC call uses the ordinary call sequence, the A register points to the parameter list. Observe that this is different from the CLOCK monitor call, where the A register points to the address of the parameter list!

```
        UPDAT=111

        LDA (NEWTIME
        MON UPDAT    % MON 111

NEWTIME,   4
          13      % Decimal:   11
          26      %            22
          11      %             9, September
        3675      %          1981
```

(To avoid converting the times to octal values, the ")DEC" mode of the MAC assembler can be used for the parameter part. In NORD-PL the "@DEC" command is used and octal values can be used preceeded by "&". Remember that the monitor call number is octal.)

In Fortran:

```
      CALL UPDAT(4,11,22,9,1981)
```

Time and calendar are set to 11.04 on September 22. 1981.

The time queue is not inspected when setting the clock absolutely. Consequently, programs scheduled through the ABSET call are treated in exactly the same way as programs scheduled by internal time (DABST, SET) and the waiting time is not affected by modification of the clock.

UPDAT does not cause the clock to stand still even if the new value is earlier the old one. The clock, including the panel clock if installed, is immediately updated with the new value.

10 RESERVING AND RELEASING DEVICES

A program does not generally receive resources automatically - an explicit request must be executed and the program may have to wait in a queue to gain control of the resource.

Even allocation of CPU capacity and memory space require explicit requests. This is usually implied by the request to start the program, but may also be explicit (e.g. FIX, FIXC).

This chapter describes requests for allocation of resources that are never implicitly granted to a program. These include all kinds of peripherals, internal and external devices.

10.1 External and internal devices

"Device" indicates a piece of hardware that can be accessed by I/O instructions, e.g. a terminal, card reader, disk, communication channel. Some devices are one way devices for input or output only. E.g. line printer, plotter (output) and paper tape reader (input).

The input, output or both parts of a two way device can reserved. If the input part is reserved, data can be read from the device; if the output part is reserved, data can be written to the device. An example of a two way device is an ordinay CRT terminal, with input from the keyboard and output to the screen.

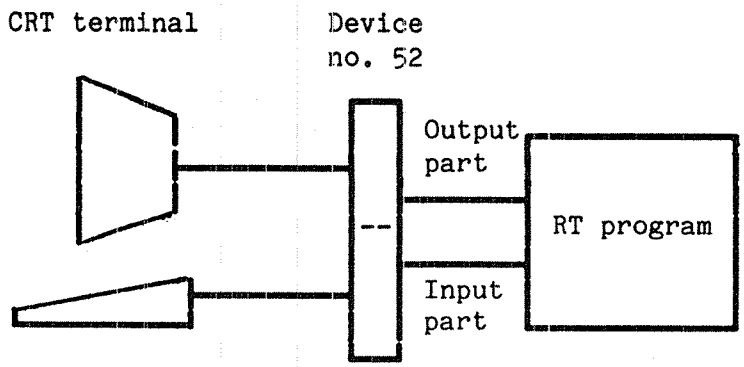


Fig. 18. Two way external device

External devices are used by a program to communicating with the "outside world". If a program wants to communicate with another program, this can be done by writing to an external device (e.g. a disk) and the other program can read from the same disk area. This involves considerable overhead and time delay.

Instead, the message can be written to a buffer in memory and read by the other program from the same buffer. This memory buffer is accessed in the same way as the disk, but is at least an order magnitude faster, is called an internal device.

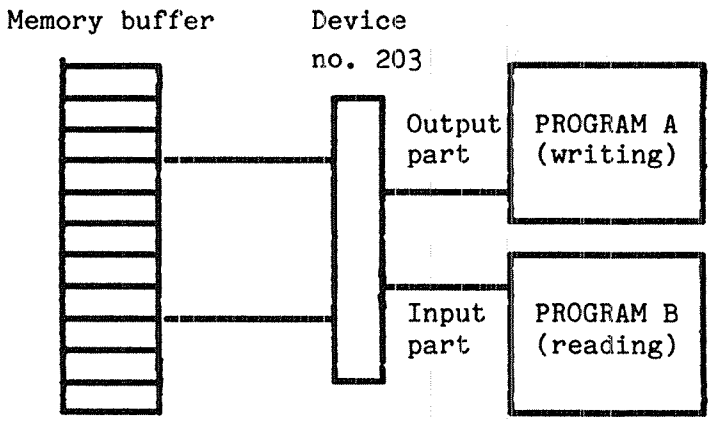


Fig. 19. Internal device

Internal devices may be simple on/off flags (semaphores), serial devices with limited capacity (byte/word oriented internal devices) or large capacity devices with dynamic buffer allocation (block oriented internal devices). The details of the different types of internal devices are given in chapters 11 and 12.

10.2 The logical device number and the datafields

Each internal or external device in a Norsk Data computer system has a logical device number (sometimes called a device number). This number is fixed and a particular type of device usually has the same device number in all Norsk Data systems. (The device also has a physical device number of no concern to the RT programmer.)

Two way devices are identified by one logical device number and when reserved the input or the output part must be specified. If both are to be reserved, two requests must be executed.

One way devices are reserved by requesting the input part (the value of the IOFLAG parameter is zero) even if the device is an output device.

Each device is described by a datafield containg information required by the driver to control the device. The driver is a routine, usually part of Sintran, controlling the device on the lowest level. Some of the fields in the datafields are common to all devices and are described in section 6.3 Others are documented in Sintran System Documentation Appendix A - Data Fields and are only relevant for experienced system programmers writing their own drivers.

Two way devices have two datafields, one for the input part, one for the output part. This allows a program to reserve one part, as the datafield is linked to the reservation queue of the RT description.

A device can only be reserved by programs executing in a ring equal to or higher than the value in the two lowest (ring) bits in the TYPRING word of its datafield. If a program in a lower ring attempts a reservation, the request is immediately rejected. The program is not put into the waiting queue, but continues execution. Any attempt to use the device causes a runtime error that terminates the program.

A program successfully reserving a device has the datafield linked into its reservation queue. The list of reserved devices is printed by the @LIST-RT-DESCRIPTION command. This list gives datafield addresses, rather than device numbers. Unlike device numbers, datafield addresses are configuration dependent.

10.3 Reserving a device

A program may request the right to use a device through the monitor call RESRV, available as a Fortran function. RESRV may be called as a subroutine, in which case the function value is discarded.

Parameters to the call are

- LDN, the logical device number of a device
- IOFLAG, indicating input or output for two way devices
- RETURN, a flag indicating return or not if unsuccessful

LDN must be found in the device number table in appendix C of SINTRAN III Reference Manual ND-60.128. If the device is a terminal, the device number can also be found from the @WHO-IS-ON or @TERMINAL-STATISTICS command executed when a user is logged in on the terminal. The device number returned by these commands is decimal, while default number base in MAC is octal.

@WHO-IS-ON

```
        49  SVEIN-ASKESTRAND
        51  P-HAUGE-PH
===>    52  REAL-TIME-GUIDE
        55  JOHN-KINSEY
        47  KNUT-NORDBYE
        60  SYSTEM
        63  KENDALL
       546  ALEX-WARMAN
       670  SYSTEM
```

The terminal currently used by user REAL-TIME-GUIDE is device number 52 decimal, 64 octal. The terminal is reserved as long as anyone is logged in. It is released after @LOGOUT, but is again reserved by the background program (BAKnn) as soon as any input is received from the terminal, even before a log in is complete.

IOFLAG should be 0 for input from the device, 1 for output to a device. For one way devices (or devices not capable of data transfer, e.g. semaphores) IOFLAG should be 0.

RETURN determines whether the program should be entered in the waiting queue if the device is not immediately available. RETURN = 0 enters the RT program in the queue and the program is passive until it is granted the right to the device.

If RETURN is 1 and the device is not available, the program is not put into the waiting queue. It continues and the A register, or Fortran function value, contains the value -1. Further information (e.g. which program has reserved the device) must be acquired through other monitor calls. If the device is available, the A register or function value is zero. A ring violation (TYPRING ring bits higher than current ring) is treated as if the device is not available and returns -1. If reservation is denied because of a ring violation, return is immediate even if RETURN is 0.

If RETURN has a value other than 0 or 1, the value returned in the A register or as a function value is always 0, regardless of whether the device was available or not.

Example:

An RT program CONTROL may control terminal 52 as long as it is not used for background processing, by waiting for it to become available while a background program is active. As soon as the user logs out, CONTROL is granted the terminal.

Input from the terminal is read by CONTROL until an ASCII ESC character is read (value: 33B), when CONTROL releases the terminal. Another ESC starts the background processor.

In this example, the characters entered are simply echoed to the terminal. In principle, any operation may be executed using the input data. Unless echo has been turned off by setting the echo mode to a negative value (MON 3, ECHOM), the terminal driver also prints input characters to the terminal. Thus in the example each character appears twice on the terminal.

After 10 seconds, CONTROL enters the waiting queue of the terminal, where it remains until the user executes a @LOGOUT command. If no other input starting the background program was received within the 10 second period, CONTROL immediately regains access to the terminal.

```
      PROGRAM CONTROL, 40
      PARAMETER (INPART=0, OUTPART=1, WAIT=0, ESC=33B)
      INTEGER CHVAL

      DO WHILE (.TRUE.)

C     Repeat until fatal machine failure....

         CALL RESRV(52,INPART,WAIT)
         CALL RESRV(52,OUTPART,WAIT)
         CHVAL =  INCH(52)
         DO WHILE (CHVAL.NE.ESC)

C         Perform any operation on input data,
C         generate any output to terminal
C         Here: simple echoing:

            CALL OUTCH(52,CHVAL)
            CHVAL = MOD(INCH(52),200B)
         ENDDO
         CALL RELES(52,INPART)
         CALL RELES(52,OUTPART)

C Wait until BAK13 has reserved the terminal, then
C enter waiting queue

         CALL HOLD(10,2)
      ENDDO
      END
```

The reservation is in MAC done by

```
      RESRV=122
      INPUT=0; OUTPUT=1
      WAIT=0

      LDA (IPAR
      MON RESRV          % Reserve input part
      JAF ERR            % Failure is indicated by -1 in Areg
      LDA (OPAR          % Success: A reg cleared
      MON RESRV          % Reserve output part
      JAF ERR

IPAR,(64                 % 52 decimal
     (INPUT
     (WAIT
OPAR,(64
     (OUTPUT
     (WAIT
```

10.4 Releasing a device

A device can be released through the RELES call. The parameters are exactly as in RESRV, but only the LDN and IOFLAG values are used. To release the input and output parts of terminal 52, execute:

```
        RELES=123

        LDA (PARI
        MON RELES
        LDA (PARO
        MON RELES

PARI,   (64         % Terminal number 52 decimal
        (0          % The input part
PARO,   (64
        (1          % The output part
```

The Fortran routine has the same parameters:

```
        CALL RELES(52,0)
        CALL RELES(52,1)
```

10.5 Reserving a device on behalf of another program

In some cases the program using a device is not the one reserving it. There are several reasons for this.

One is when a program is written to perform some kind of service on a number of devices, but the selection of which device to work on is done by another program. Another case is where a device is handled by several programs and the device is passed from one program to the next.

A device may be reserved for a program through a command:

```
Sintran command: @PRSRV <log. unit> <input/output> <RT name>
Monitor call:    MON 124  % PRSRV
```

Command parameters:

<log. unit> - decimal logical device number of the device to be reserved

<input/output> - a 1 indicates the output part, a 0 the input part of a two way device

<RT name> - the name or RT description address of the program which should be granted access to the device

The argument list is in MAC pointed to by the A register and the status of the reservation is returned in the A register; if the reservation was successful, the A register is zero, otherwise it is -1. The most common cause of failure is that the device was already reserved or that the TYPRING bits in the data field do not permit reservation. Unsuccessful completion of the @PRSRV Sintran command does not give an error message.

The ring bits of the program receiving the device determine whether reservation is permitted or not. The priority of the reserving program determines the position in the waiting queue if the device is not immediately available and the reserving program is put into the waiting queue (there is no way to request return if the device is not immediately available).

Below is a program fragment where the program SENDER is given the output part of an internal device, number 202B. The present program acquires the input part to receive messages from SENDER. It is assumed that SENDER is in an RTWT state and will fetch data forming the messages from an external source when it is restarted.

```
        PRSRV=124; RT=100; RESRV=122

        LDA (SND
        MON PRSRV           % Output part reserved for SENDER
        JAF FAIL            % Non-zero if not available
        LDA (MYSLF
        MON RESRV           % Input part for this program
        JAF FAIL
        LDA (SNDPR
        MON RT              % Start SENDER
         . . .
FAIL,                       % Error handling routine


SND,    (202                % Internal device
        (1                  % Output part
SNDPR,  (SENDER
MYSLF,  (202                % Same internal device
        (0                  % Input part
        (1                  % Return with A = -1 if failure
```

10.6 Forcing a program to release a device

Deadlock situations are often caused by the failure of program to release a device. They may be fatal to the entire system, but if not, another program or an operator can resolve the deadlock by forcing the program monopolizing a device to release it. This may be fatal to the program, if it attempts to access the device it no longer has reserved, so it should be used with care.

Less dramatically, a high level administrative program may release a device from one program to give it to another one through PRSRV. Even in this case, care should be exercised to ensure that operations on the device are properly completed first.

The PRLS call is available as a Sintran command and monitor call:

```
Sintran command: @PRLS <log. unit> <input/output>
Monitor call:    MON 125  % PRLS
```

Command parameters:

<log. unit> - decimal logical device number of the device to be released

<input/output> - 1 indicates the output part, 0 the input part of a two way device

The list of parameters to MON PRLS is pointed at by the A register. The SENDER program in the previous section may, when the communication is complete, go into an RTWT state waiting for the next activation. This does not release the reserved device. To force SENDER to give up the internal device, the instructions executed by the receiving program could be:

```
        PRLS=125

        LDA (SND
        MON PRLS
         ...

SND,    (202        % Internal device
        (1          % Output part
```

Equivalent Fortran code:

```
        CALL PRLS(202B, 1)
```

Note that the receiving program still reserves the input part of the device. Release of input and output parts are independent.

The PRSRV and PRLS calls are often used together to take a device from one program to give it to another one. In certain cases this fails:

As soon as the device is released, forceibly or voluntarily, it is granted to the first program in the waiting queue. This might happen before the program executing the PRLS has had time to grant the device by PRSRV to the program it intended and the reservation fails.

This probability of this can be reduced by giving the program executing the PRLS/PRSRV a higher priority than any program in the waiting queue. This adjustment can be temporary and reset as soon as the reservation is successful. Even this does not guarantee that no problems occur.

If the terminal used by a background program is released through PRLS, the background program is terminated and the user logged out, unless the command is executed from the released terminal.

OBSERVE:

Care should be taken when a device is forcibly taken from a program with the intention of giving it back to the program later. If the buffer (input or output) is not empty, the owner of the device may be hanging in an IOWAIT. If the transfer finishes while the device is "borrowed", the owner is not properly restarted.

The probability of this happening is reduced if the PRLS is delayed until the buffer is empty. This is checked through the OSIZE call (section 13.12).

10.7 Reserving a directory

File system maintainance systems may need exclusive access to an entire dircetory. This can be granted, provided that the directory is entered and no files are opened on the directory (if a main directory, no user may be entered).

The directory is reserved by the REDIR call, and released by the RLDIR call. The T register must hold the directory index of the directory to be reserved. The @LIST-DIRECTORIES-ENTERED command will list the directories in order of increasing directory index (first directory=0), but be aware that if no directory is entered on a device, an LF only will be printed for that entry in the directory table. A program may use FOBJN (MON 273) reading the directory (and also user and object) indexes of any file on the directory.

```
REDIR=246
RLDIR=247
PCTWO=1

SAT PCTWO    % Reserve the PACK-TWO directory
MON REDIR

%  Use directory

SAT PCTWO
MON RLDIR
```

These calls are available in Sintran version H and later. They are also available as commands. See SINTRAN III Reference Manual ND-60.12.

10.8 Reserving devices through Sintran commands

A program usually reserves the devices it needs through monitor calls. Sometimes a terminal operator wishes to request use of a device not normally continuously reserved by the terminal background program.

An example of this is a magnetic tape station: after mounting a tape the owner of the tape wants to keep others from using the station. To prevent it the tape drive is be reserved for the terminal user.

10.8.1 Reserving a file for the users terminal

To prevent other users accessing the floppy, a user can reserve the peripheral file associated with the floppy device while he is copying files to or from the floppy.

If a user wants to copy several files to the magnetic tape without interruption, the peripheral file for the magnetic tape may be reserved for the user's terminal.

Programs communicating through files are dependent on having full control over the file. Opening the file reserves the device; if there is a time lag between one program's closing and the next program's opening of the file, another program might "steal" the file as soon as it is closed.

A file is reserved by the command

@RESERVE-FILE <file name>

Command parameter:

<file name> - name of a peripheral file

If the file is not a peripheral file, no action is taken, however, the command is legal for all files and no error message is issued.

The file may later be released through the command

@RELEASE-FILE <file name>

The parameter is the same as for @RESERVE-FILE

(It is possible to run a SINTRAN III installation with no peripheral file name for the floppy device. However, in most systems the peripheral files are created when the system is installed. The file name of the floppy drive is used when a file is written to the floppy disk without the use of a directory and when the @DEVICE-FUNCTION commands are used, e.g. to format a floppy disk.)

10.8.2 Reserving a device unit for the user's terminal

@RESERVE-DEVICE-UNIT <device name> (<unit>) (<F/R>)

Command parameters:

<device name> - the name of the mass storage device controller to which the device is connected. The standard names are found in appendix C of SINTRAN III Reference Manual ND-60.128.

<unit> - the unit number of the device, if more than one device is connected to the controller.

<F/R> - only to be used with cartridge disks. F indicates the fixed disk pack, R indicates the removable disk pack.

The device name is used rather than the peripheral file name. @RESERVE-DEVICE-UNIT prevents directories from being entered on the device and is used before starting a program using the floppy drive in a nonstandard way. The program may write any data on the device without using the file system for logically structuring the disk.

No user, including the user reserving the device unit, can enter a directory on the drive. The command prohibits any file system operations on the disk and its datafield is entered in the BRESLINK queue of the background program.

Before a directory can be entered, the user who reserved the device unit must log out (releasing any devices reserved by his background program) or execute the command

@RELEASE-DEVICE-UNIT <device name> (<unit>) (<F/R>)

The parameters are the same as for @RESERVE-DEVICE-UNIT.

These two commands are permitted for all users, but the command releasing the device unit must be executed at the terminal where the reservation was made.

10.9 Determining who has reserved a device

In the analysis of a deadlock situation, it is vital to find out which programs are occupying the various devices involved. The commands required to find the relevant information depend on the kind of reservation.

10.9.1 A file reserved through @RESERVE-FILE

The reserver of a file may be identified by the Sintran command

@WHERE-IS-FILE <file name>

This command can be used for any file and is available to any user. It identifies the user by name and terminal number. If the file is opened by by an RT program, the respose is

<file name> RESERVED BY USER RT ON TERMINAL 1

if the RT program is terminal independent. This does not imply that there is anyone logged in as user RT on the console.

If the file is a peripheral file reserved by a RESRV or PRSRV call, the response is

<file name> RESERVED BY RT PROGRAM <RT name>

10.9.2 A device identified by a logical device number

```
@LIST-DEVICE <log unit> <input/output>
```

This command expects a decimal logical device number and 0 indicating input or 1 indicating output. The reserving program is identified by name. If the device is a terminal, the reserving program is usually a background program (BAKnn).

If programs are in the waiting queue of the device, they are identified by name:

```
@LIST-DEVICE 52 0
RESERVED BY: BAK13
WAITING RT-PROGRAMS:
CONTROL
```

The response NOT RESERVED indicates that any program requesting the device will immediately be granted access.

A program may use the monitor call WHDEV (MON 140) to determine who has reserved a device. The RT description address of the reserving program is returned in the A register or as a Fortran function value.

Example (tape punch is logical unit 2):

```
      PARAMETER(PUNCH = 2)
      INTEGER WHDEV

      IPROG=WHDEV(PUNCH,0)
      IF (IPROG.EQ.0) THEN
C        reserve and use paper tape punch
      ENDIF
```

IPROG receives the RT program description address if the device is reserved (0 if free) and the program can take different actions depending whether it is a background or RT program.

10.9.3 A device identified by a datafield address

In a deadlock a program may be "eternally" waiting for a device. The RT description, written by @LIST-RT-DESCRIPTION, continuously reports: WAITING FOR nnnnn, where nnnnn is a datafield address.

If the logical unit number corresponding to the datafield is unknown, the RT description address of the reserving program can be found by inspecting the RTRES location of the datafield. User SYSTEM can inspect resident memory through @LOOK-AT RESIDENT; the first location following the data field address is the RTRES field (see section 6.3) containing the RT description address of the reserving program.

After the RT description address is found, the name of the RT program can be found by the command @GET-RT-NAME.

Example:

```
@LIST-RT-DESCR CONTROL
 RING:0 PRIORITY:     40
LAST STARTED:    2 MINS   48 SECS
START ADDRESS:      0, SEGMENTS:      0    322
P=    226
X=    162
T=      3
A=      0
D=    225
L=     14
S=      0
B=    344
WAITING FOR:  23126
ACTUAL SEGM.:      0    322
@LOOK-AT RES
READY:
23126/  40417
  40417 .
-END

@GET-RT-NAME 40417
BAK13
```

The device 23126 (terminal 52 in this system) is reserved by the background program BAK13.

A datafield address does not necessarily correspond to a logical unit number, but may be an internal device or a semaphore used by the operating system, not available to users. Even for those datafields that do correspond to an external or device, there is no general mechanism to translate a datafield address to a logical unit number.

If a programmer has the listing of Sintran PART-TWO (the configuration dependent part), he can look up the datafield address here and in many cases will be able to identify the device. Reading any part of the Sintran listing requires some experience with NORD-PL and familiarity with the configuration.

10.10 Reservation in SINTRAN III vs. the "Dijkstra semaphore"

Computer literature often discusses binary semaphores as defined by A.W.Dijkstra and a multitude of algorithms for building higher level synchronization tools based on this concept are available.

Essentially, reservation of devices other than semaphores follows the same pattern as reservation of semaphores. Comments below regarding semaphores apply to all internal and external resources that can be reserved in a SINTRAN III system.

Advanced programmers should be aware that the reservation of devices (including semaphores) does not exactly follow the rules defined for the Dijkstra semaphore.

The algorithm for the reservation of a Dijkstra semaphore is as follows:

```
IF REQUESTED RESOURCE AVAILABLE
THEN
   ALLOCATE RESOURCE TO REQUESTING PROGRAM
ELSE
   ENTER REQUESTING PROGRAM INTO WAITING QUEUE
ENDIF
```

The SINTRAN semaphore is reserved through the following algorithm:

```
IF REQUESTED RESOURCE AVAILABLE
THEN
   ALLOCATE RESOURCE TO REQUESTING PROGRAM
ELSIF  OWNER >< REQUESTING PROGRAM
THEN
   ENTER REQUESTING PROGRAM INTO WAITING QUEUE
ENDIF
```

The only difference between the two is when a program tries to reserve a resource it has already reserved. The Dijkstra semaphore causes the program to deadlock; it enters the waiting queue, waiting for the device to be released, but as it is passive in the queue, it can never complete its operations on the device to release it.

Reservation of a SINTRAN semaphore which the requesting program has already reserved is effectively ignored; the program can therefore continue execution without causing a deadlock.

This has two side effects:

Certain statements ("invariants") about the semaphore do not necessarily hold true. For a Dijkstra semaphore, the number of successfully completed WAIT operations (RESRV calls) executed on a device cannot exceed the number of SIGNAL operations (RELES calls) plus one. Repeated reservations of a Sintran semaphore violate this invariant, while according to the rules of the Dijkstra semaphore a deadlock should occur.

Even though several RESRV calls have been executed, only one RELES call is required to release the device. If the program is not aware that the device reserved is the same as one already reserved, it may assume that the "second" device is still available after the first one has been released.

To avoid some of these problems and prevent confusion,

- a device should not, if possible, be accessed as a peripheral file and a device by the same program

- if a peripheral device is created for an internal device, separate files should be made for the input and output parts (a file with R access only for the input part and one with W or WA access for the output part)

- in critical program systems, the program should maintain a list of the device numbers reserved and every time a peripheral file is opened or a device number obtained from another program, it should be checked against this list.

10.11 Obtaining information about devices

There is no general mechanism for reading the data field of a device. However, the device type and a set of attributes regarding the permitted operations on the device may be obtained by the GDEVT call. Programs may be designed to handle arbitrary device types, determining the appropriate device handling from the type and attributes.

```
        GDEVT=263

        LDT INDEV     % T = device number
        SAA 0         % A = 0: input, A = 1: output
        MON GDEVT
        JMP ERR       % Error return, A = error code
        COPY SD DA    % 32 attribute bits returned in AD
        AND (10       % Check bit 3: block calls allowed
        JAZ BYOUT     % Not set, use byte IO
         . . .        % Block calls permitted

BYOUT,  . . .         % Byte IO handling
```

On return the T register will contain the device type:

0 = unspecified
1 = terminal
2 = background access device (BAD)
3 = communication channel (ND-net etc.)
4 = internal block device
5 = floppy disk
6 = magnetic tape
7 = mass storage file

The AD register pair will contain 32 attribute bits, of which 6 are currently used:

bit 0: INBT/OUTBT allowed
bit 1: CONCT allowed
bit 2: IOSET allowed
bit 3: block IO calls allowed
bit 4: clear device routine available
bit 5: no reservation of device necessary

If a peripheral file is opened through MON OPEN, both its open file number and its logical device number can be obtained through MON FOPFN:

```
        FOPFN=257

        SAX PLOT
        SAA EMPTY
        MON FOPFN
        JMP ERR       % Error return: A = error code
        STT OPNO      % T = open file number
        STA OPCOD     % A = open code
        COPY SD DA    % D = peripheral device no
        STA DEVN


PLOT,   'PLOTTER-1'
EMPTY,  ''
```

The open code in the A register has the value 0 if the file is opened for read, 1 if opened for write and the value 2 if opened for both read and write. The file must be opened for RT programs.

These monitor calls are available in Sintran version H and later.

11 SEMAPHORES

The simplest form of communication is a binary signal, it is either on or off. E.g. a numeric value of 0 or 1, a resource is available or unavailable, a process has completed or it has not, etc.

Such signals are transmitted between programs by semaphores. By convention, the terms used to describe the states of a semaphore are reserved or free, but the interpretation of these terms is up to the accessing programs.

There are two major uses of semaphores:

> As a timing signal. Program "A" reserves a semaphore while it is performing its operations. Another program "B" which requires that A completes before proceeding must wait for the semaphore to become free.
>
> As a reservation flag. Program "A" working on a resource, e.g. a memory buffer, must, in order to guarantee consistency of a data structure, prevent other programs from accessing that same memory buffer. It does that by reserving a semaphore and expects all programs competing for access to the buffer to wait for the semaphore to become free.

Semaphores are used as timing signals to prevent deadlocks. SINTRAN does not provide concurrent reservation of several resources (files, devices etc.) and a sequence of reservations may be incomplete if one of the resources was unavailable or the sequence was interrupted by another program requiring one or more of the yet unreserved resources. A group of devices can be protected by a semaphore, preventing a program from reserving one device if they are not all available.

Resources identified through a logical device number (peripherals, internal devices, files) are protected from concurrent access by the operating system. Semaphore protection is required only where resource sharing is beyond the control of the operating system, such as use of data in a common segment or execution of a nonreentrant routine.

11.1 Semaphores and protocols

The use of semaphores is completely under control of user programs and SINTRAN does not guarantee consistent use of the semaphores beyond that of the reserving the semaphores themselves.

Any program may ignore the semaphores protecting a common resource. If the resource is not a device already reserved by another program (and thereby protected from concurrent accesses by the operating system), there is nothing to stop the program from reserving, accessing or modifying data in the protected group.

It is up to a programmer to ensure that all system components respect the rules enforced by the semaphores. Techniques exist to build higher level synchronization tools and some of these are illustrated in the examples. The semaphore should be considered a basic building block for protection mechanisms and synchronization tools tailored to the particular application.

However there is no way to guarantee that programs external to the system do not violate these protocols.

11.2 Access to semaphores in Sintran

A semaphore is identified by a logical unit number between 300B and 377B. The number of semaphores available is a system generation parameter and is generally between 5 and 20. Default number of semaphores when ordering SINTRAN is 5.

These semaphores are all available to user programs. In addition, the operating system uses a number of semaphores internally to protect data structures used by the RT monitor; these cannot be used by user RT programs.

A semaphore is reserved and released by the same monitor calls as other devices:

RESRV - reserve a semaphore
RELES - release (free) semaphore
PRSRV - reserve a semaphore on behalf of another program
PRLS - force a program to release a semaphore

Like any other device, a semaphore has a waiting queue and a program which unsuccessfully attempts to reserve a semaphore is put in its waiting queue.

11.3 Example: access conflicts causing inconsistent data structure

This example illustrates how access conflicts can cause an inconsistent data structure if several programs access shared variables:

Consider a list of elements, indentified by alphabetic characters:

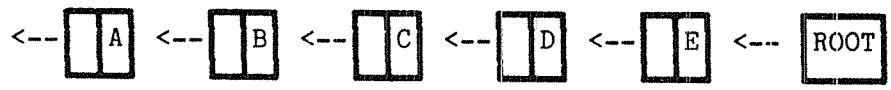


Fig. 20. Linked list

The "<--" is a pointer - the address of another element in the list and is termed the "next-pointer".

A service program MOVE moves a specified entry ELEM from its current position in the list to another position. The new position is identified by the list element AFT after which ELEM should be inserted (ELEM is inserted to the left of AFT in the figure above).

Activation of MOVE requests "B" to be moved after "D". This is performed by unlinking the element "B" (using an auxillary pointer as a temporary reference to "B") and setting the next-pointer of "B" to the same element as "D"'s next-pointer

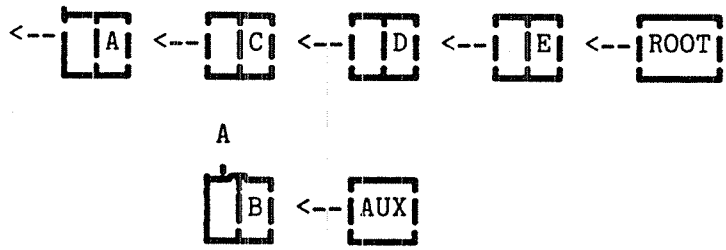


Fig. 21. B has been unlinked

The intention is now to set "D"'s next-pointer to point to "B", completing the insertion. However, this is interrupted by another program.

The interrupting program is also one modifying the list and it moves "D" to after "A". The picture is now:

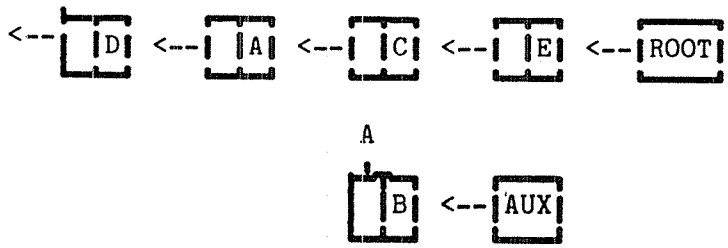


Fig. 22. D is moved

After this operation, MOVE takes over again and completes its operations by setting "D"'s pointer to point to B. The temporary auxillary pointer is no longer needed. The data structure is now

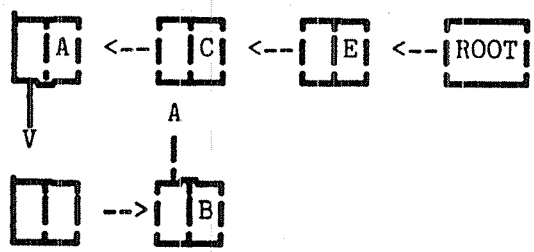


Fig. 23. An unintended loop

Any routine that performs an operation "on every element to the end of the list" goes into an infinite loop. Wherever a test is performed to see if there is a "next" element or not, the answer is "yes". The unintended result was because of the interrupt midway through the operations.

11.4 A solution using semaphores

If all devices accessing the data structure headed by ROOT must reserve semaphore 301B before they modify it, each update of the data structure can be completed before another update starts. This makes it far easier for each program in the system to guarantee data consistency.

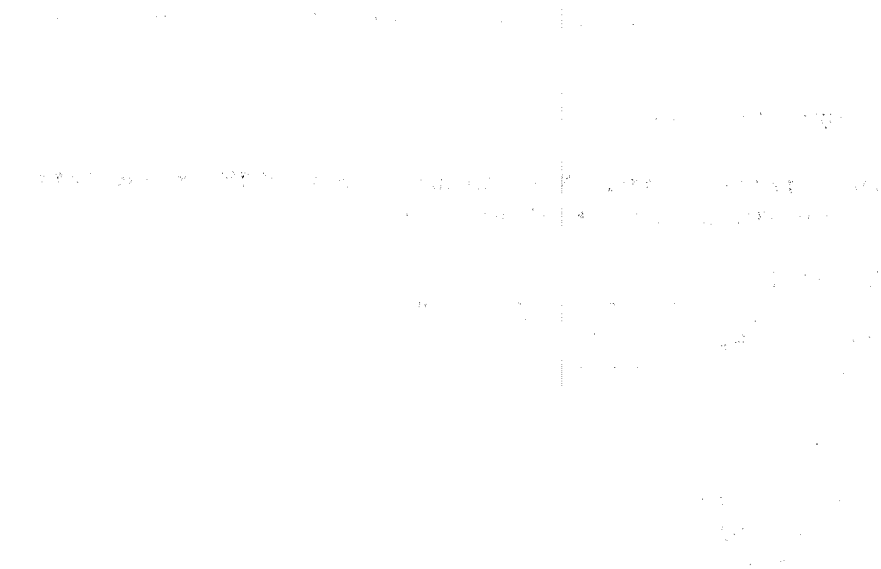
The program solution is written in Fortran. As record and pointer concepts are not present in Fortran, a list must be simulated by a two-dimensional array, where the next-pointer is the array index of the next element rather than its memory address.

The array LIST is declared as a 2 by 5 array, limiting the number of elements in the list to 5; index 0 is the ROOT. The two locations of element "n" are the next-pointer as (1,n) and the ASCII value of the alphabetic identifier as (2,n). In a more realistic situation one would add various data values as (3,n), (4,n) etc.

```
      PROGRAM MOVE, 50

      PARAMETER (NXTPTR=1,VAL=2,ROOT=0)

      INTEGER AUX, TEMP, LIST(2,0:5), ELEM, AFT
      COMMON /SHARED/LIST
      COMMON /ARGUMENT/ELEM,AFT

C     Reserve data structure:

      CALL RESRV(301B,0,0)

      AUX = LIST(NXTPTR,ROOT)

C     Unlink an element:

      DO WHILE (LIST(VAL,AUX).NE.ARG)
         TEMP = AUX
         AUX = LIST(NXTPTR,AUX)
      ENDDO

C     "Splice" the chain where the element was removed

      LIST(NXTPTR,PREV) = LIST(NXTPTR,AUX)

C     Find where the element should go into list

      TEMP = LIST(NXTPTR,ROOT)
      DO WHILE (LIST(VAL,TEMP).NE.AFT)
         TEMP = LIST(NXTPTR,TEMP)
      ENDDO

C     Insert element into list:

      LIST(NXTPTR,AUX)  = LIST(NXTPTR,TEMP)
      LIST(NXTPTR,TEMP) = LIST(NXTPTR,AUX)

C     Free the data structure for use by others:

      CALL RELES(301B,0)

      END
```

A program requesting "B" to be moved after "D" would look like:

```
      PROGRAM PRG, 40

C     Lower priority than MOVE assures that MOVE is complete
C     before PRG continues after call

      EXTERNAL MOVE
      INTEGER LIST(2,0:5), ELEM, AFT
      COMMON /SHARED/LIST
      COMMON /ARGUMENT/ELEM,AFT

      . . .

      ELEM = ICHAR('B')
      AFT  = ICHAR('D')
      CALL RT(MOVE)

      ...

      END
```



12 INTERPROGRAM DATA EXCHANGE

Cooperating programs need some way of exchanging data, such as results from computations, error status values, input which is preprocessed by another program or a record fetched from a database.

There are several mechanisms available for data transfer. The most important criterion for selecting one is the transmission speed required, In general the higher the transmission speed, the higher the cost in terms of system resources.

The methods of communication available are:

RTCOMMON
Shared segment
Internal devices (byte/word oriented)
Internal devices (block oriented)
XMSG
Files

This chapter describes use of RTCOMMON, internal devices and shared segments. XMSG, a more general communication system, is treated in chapter 20 and use of files is the topic of chapter 13.

Semaphores (see the previous chapter) are used for communication of timing signals and reservation flags, but are not themselves data communication channels.

Internal devices (byte, word and block oriented) and shared segments can be used within one CPU only; SINTRAN III has no mechanisms for coordinating the Sintran resident areas and segment handling of two separate systems. For communication with other systems, RTCOMMON can be used if the CPUs involved has access to the same (multiport) memory. XMSG can be used whether the two (or more) machines have common memory or not.

Shared segment is treated as a special case in the ND-500 monitor. A ND-500 computer may access a segment in the ND-100 if the segment is fixed in a contiguous area of memory (FIXC call - see section 8.2.5).

12.1 RTCOMMON

An area of physical memory, RTCOMMON, is fixed; the contents of these locations are never removed to make room for other data. In many respects this area behaves as a fixed segment that is never unfixed, hence its contents are never swapped back. After a warm-start, the contents of RTCOMMON is unchanged. The user should note that RTCOMMON is not to be looked upon as a segment.

Locations in RTCOMMON are addressed in the same way as other variables in the program and operated on by load and store instructions. In a high level language such as Fortran, this corresponds to assigning a value to a variable or using the value of a variable within the RTCOMMON area.

All RT programs running in ring 1 or above and using page table 1, address the RTCOMMON area in the uppermost locations in the logical address space. The same physical locations are addressed by all RT programs using PT1 and modifications in RTCOMMON have an immediate effect for all programs using the modified locations.

Transmission is immediate; there is no delay, regardless of priorities, system load or load on I/O channels.

Programs running in ring 0 cannot access the RTCOMMON area; a protect violation interrupt will be generated. Nor do programs using page tables 0, 2 or 3; they have their own private pages and can use the entire logical address space including the uppermost locations without concern for other programs. A program may have its code on a segment on another page table than 1, and address RTCOMMON by the alternative page table mechanism.

12.1.1 Access to RTCOMMON

Data to be placed in RTCOMMON is declared in a named Fortran common block. Before the file containing the first occurence of this block is loaded, the label must be defined as an RTCOMMON label through the RT loader command *SET-RTCOMMON.

Example:

Fortran program:

```
PROGRAM START,50
INTEGER FLAG(10)
COMMON/FLAGS/FLAG

FLAG(1)= 1
CALL RT(CONTINUE)
END
```

Load procedure:

```
@RT-LOADER
*NEW-SEGMENT 312,,,,,
*SET-RTCOMMON FLAGS
*NREENTRANT-LOAD START,,
*END-LOAD
```

The integer array in FLAGS is available to all programs on page table 1 and any program can modify it. Programs loaded at any time after the *SET-RTCOMMON command has been executed may refer to the label as an external symbol. Remember that the common block name, rather than the names of the symbols in this block, is the external definition!

The symbols defined in RTCOMMON may be listed by the command *WRITE-COMMON-LABELS. Any global common labels can be listed by this command - the segment number of RTCOMMON labels is zero. The three numerical values returned are the address, the segment number (equal to zero for RTCOMMON) and the size of the common block.

```
*WRITE-COMMON-LABELS
OUTPUT FILE: TERMINAL

   AB       172000    0   12

*
```

The common block AB is located at address 172000 and is 12B (10 decimal) words.

12.1.2 Inspecting RTCOMMON variables through @LOOK-AT

Individual locations in RTCOMMON can be inspected and modified, provided that their addresses are known. @LOOK-AT RTCOMMON accepts numeric arguments only; the addresses must be found by *WRITE-COMMON-LABELS and possibly adding a displacment within the common block.

The syntax of @LOOK-AT RTCOMMON subcommands is the same as for other @LOOK-AT commands: a slash (/) indicates that the preceeding value is an address, (CR) that the next location should be displayed, a dot (.) terminates the command.

```
@LOOK-AT RTCOMMON
READY:
172000/      5217   (cr)
0                   .
-END
@
```

12.1.3 The size of RTCOMMON

When ordering a Sintran system, the amount of RTCOMMON space must be specified. The size of RTCOMMON is a system generation parameter and is usually fairly small, e.g. eight pages (address range 160000B to 177777B). If not requested, no RTCOMMON is reserved, in which case RT programs cannot use this as a means of communication.

Although the RTCOMMON size is set at system generation time, another area to be used as RTCOMMON can be defined by user SYSTEM through the @SINTRAN-SERVICE-PROGRAM (not available to user RT):

```
@SINTRAN-SERVICE-PROGRAM
*DEFINE-RTCOMMON-SIZE
RTCOMMON SIZE: 4
FIRST PHYSICAL PAGE(OCT): 374
IMAGE? Y
SAVE-AREA? Y
```

"Resident" may not be modified; the system must be restarted ("warm start") before the change has any effect. In this example, 4 pages (8 kbyte) occupying the uppermost pages of a 1/2 megabyte of memory was reserved for RTCOMMON. These 4 pages are in addition to what was specified at system generation.

Default physical address is the upper end of memory. In the example above, default could have been used.

The specified size may not exceed the amount of RTCOMMON at system generation time by more than 8 pages (16 kilobytes). If the *DEFINE-RT-COMMON is used more than once, the last definition applies and all previously defined areas are cancelled.

RTCOMMON is usually located in the upper part of physical memory. In certain cases it is necessary to use another area, usually due to special I/O devices using the RTCOMMON area. If an ND-500 communicates with an ND-100 through RTCOMMON, the RTCOMMON area must be contiguous. In other words, the area defined by *DEFINE-RT-COMMON must be adjecent to the area defined at system generation time.

The upper part of PT1 always points to RTCOMMON, thus a large RTCOMMON limits the maximum area for free use by PT1 programs.

12.1.4 Concurrent access to RTCOMMON

When running a one CPU system with local memory only, no concurrent access to a single RTCOMMON location occurs, because only one process is active at a time.

If several CPUs or DMA devices have access to a multiport memory there may be two or more devices attempting to read or write the same location at the same moment. Hardware circuits in the multiport (called an "arbiter") ensures that the devices are granted access one by one. Each write operation overwrites the previous value of the location. The last writer determines the final value.

Thus, access to RTCOMMON should be controlled through a semaphore or other synchronizing mechanisms, to prevent access conflicts or inconsistent data.

12.1.5 Example: using an RTCOMMON variable as a semaphore

By using certain machine instructions, RTCOMMON locations can be used to build synchronizing mechanisms.

The MIN machine instruction increments a memory location by 1 and if the incremented result is 0 the next instruction is skipped. All access to memory is done in one uninterruptible cycle. If several CPUs perform a MIN on a location, every increment performed and the final result is the starting value plus the number of increments performed.

A simple semaphore without an associated waiting queue is often sufficient to mark a data area as available or in use. If the area is in use, a new attempt to reserve the area is made after a one second pause.

The RTCOMMON variable FREE has the value -1 if the area is available, 0 if the area is in use.

```
)9BEG
)9ENT FREE

FREE, 0

)9END

)9BEG
)9ENT RESERVE, RELEASE
)9EXT FREE

      STZ FREE            % Restore "reserved" flag
      LDA (1SEC           % Wait for release
      MON HOLD
RESERVE,
      MIN FREE            % Increment FREE from -1 to 0
      JMP *-4             % No - FREE was not -1
      EXIT                % Resource was reserved

RELEASE,
      SAA -1
      STA FREE
      EXIT
)9END
```

The symbol FREE must be declared as an RTCOMMON symbol by the #SET-RTCOMMON command. FREE is made a separate BRF module (delimited by 9BEG and 9END) in order to load FREE alone, without including the code part, to RTCOMMON.

Even executable code may be loaded to RTCOMMON. Compared to a fixed segment startup time is shorter because no loading of page tables is required. However, available space usually prohibits programs to reside entirely in RTCOMMON.

12.2 Byte oriented internal devices

An internal device is a communication channel operated on as a file, through INBT and OUTBT monitor calls. But while the information written to a file usually is stored on a disk, the information written to an internal device is stored in a memory buffer, where another RT program may read from the same internal device.

Because of the buffering, there may be a delay between the writing and the reading operation. The device operates as a mailbox, where a message is stored until needed and all messages are received in the same sequence as they are sent. In a byte oriented internal device, each message is one byte. A standard modification (patch) may change the size of each message to a 16 bit word (word oriented internal device).

12.2.1 The number of internal devices

The number of byte oriented internal devices is a system generation parameter, but unless otherwise specified, the system has two devices. The maximum number possible is 40B (32).

If the configuration includes the Sibas data base system, each Sibas process requires two internal devices for its own use. Other subsystems may also need internal devices.

12.2.2 The ring buffer

The size of the buffer is limited, by default 64 words (128 bytes), limiting the amount of data that can be written before any of it is read. A read removes a byte, freeing the space for future write operations. This kind of buffer is called a ring buffer and is displayed as a circular chain and two pointers, the write pointer, pointing to the next element to be written and the read pointer, pointing to the next element to be read.

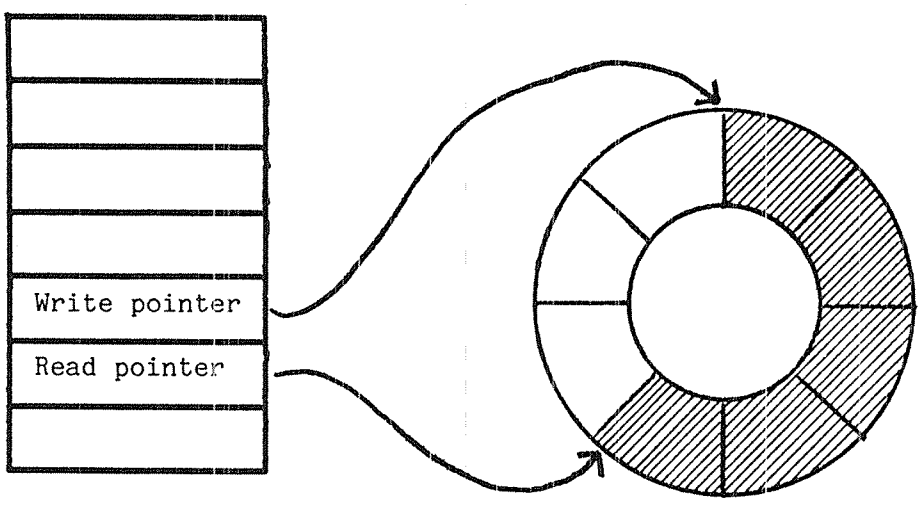


Fig. 24. Ring buffer and datafield

The area between the read pointer and the write pointer represents the

amount of data currently in the buffer. If the read pointer catches up with the write pointer, the buffer is empty; there is nothing more to read. If the write pointer gets one round ahead of the read pointer and catches up with it from behind, the buffer is full; there is no more space for more writes until a read operation takes place.

If a program attempts a read operation while the buffer is empty, the program is (normally) set in an IOWAIT state, where it remains until data is written to the device. Similarly, if a program attempts a write while the buffer is full, the program is (normally) set in IOWAIT.

12.2.3 Reserving an internal device

Internal devices are reserved in the same way as external devices (see chapter 13). Each has a device number in the range 200B-237B (limited by the configuration). The device has an input (read) and an output (write) part, each reserved separately and usually by different programs.

```
PROGRAM A, 40

CALL RESERV(203B,1,0)
....


PROGRAM B, 40

CALL RESERV(203B,0,0)
```

12.2.4 Reading and writing

After the device has been reserved, input or output may be performed through the INBT and OUTBT monitor calls (MON 1 and MON 2).

The multiple-byte monitor calls B4INW, B8INB, M8OUT and B8OUT can also be used for reading from or writing to an internal device. These calls reduce overhead somewhat and may be used to ensure that the recipient has the entire message available (or that the entire message is read) in one operation, uninterruptible by process switching. These monitor calls are not available from Fortran.

In high level languages like Fortran, the ordinary write statements for sequential output are used. In Fortran, these are the READ and WRITE statement:

```
      PROGRAM A, 40
      CALL RESERV(203B,1,0)
      IX = 25
      WRITE(203B,100) IX
100   FORMAT(I10)
      END

      PROGRAM B, 40
      CALL RESERV(203B,0,0)
      READ(203B,200) IX
200   FORMAT(I10)
      END
```

All characters are transmitted through the internal device, including position 1 in the line, without regard to printer control codes.

Fortran programs can also call the INCH and OUTCH subroutines to write individual bytes, specified by their integer values. This is commonly used if the transmitted data does not represent alphanumeric information. For example may a separate command processing program read a command, look it up in a command table and transfer the index in table to another program, rather than the command string itself:

```
      CALL RT(CMDPROC)

C Reserve input part to read command index
C returned from command processor

      CALL RESRV(201B,0,1)
      IXCMD=INCH(201B)
      GOTO (100,200,300,400,500) IXCMD
```

12.2.5 Reading the amount of data in the buffer

A program may want to see how far a companion program has progressed in reading the buffer used for exchanging data between the two. It may want to check whether there is still room in the buffer for another data byte to be written, or another byte available for read, before the operation is started. This may prevent the program from entering a waiting state. (This can also be done by requesting NOWAIT mode, but for a number of reasons it may be more desirable to check the space available in the buffer explicitly.)

The number of bytes currently in the ring buffer is read through ISIZE (MON 66). As there is one common buffer for the input and output part, the number of bytes that may be written before the buffer is full is found by subtracting ISIZE from the size of the buffer (default 100B words). The OSIZE call, used to read the number of unused bytes in the output buffer of an external device, is meaningless with respect to internal devices (the common buffer is a part of the input datafield).

ISIZE = 0 implies that a program attempting to read from the device enters a waiting state (buffer empty). Otherwise, the number of bytes still available for read is returned.

ISIZE = buffer size implies that a program attempting to write enters a waiting state (buffer full). The difference between ISIZE and the buffer size gives the number of bytes that may be written without entering a waiting state.

By requesting NOWAIT mode (described in connection with files, section 13.11.1), entering a waiting state is prevented even if the buffer is empty on read or full on write.

The device need not be reserved prior to the call. The device number is loaded to the T register or is the function argument in Fortran. The number of bytes available is returned in the A register or as a Fortran function value:

```
ISIZE=66
INBT=1

LDT (203     % Internal device 203B
MON ISIZE
JAZ DOMORE   % Nothing to read yet...
MON INBT     % T still contains device no
   .         % Handle input
```

The call is available as a Fortran integer function:

```
IF (ISIZE(203B).EQ.0) CALL DOMORE
```

The ISIZE call is available on external devices and files as well as internal and is useful to check e.g. whether any input has been entered from a terminal (see section 13.12). It is also available from background programs. (Background users should be aware that the current terminal can not be identified by unit number 1, but must be found through RSIO, MON 143.)

12.2.6 **Clearing the device buffer**

The program reading from or writing to a device can reset the number of bytes buffered to zero by the CIBUF (MON 13) call.

The CIBUF call is functionally equivalent to reading and immediately discarding all input available in the buffer at the time of the call. This is legal only if the input part of the device has been reserved.

In assembler, the T register must contain the device number and the A register contains an error code, if any. In case of error, return is to the first location following the MON instruction, while ordinary return is to the second instruction following the call ("skip return").

12.2.7 Changing the buffer size

The default buffer size of an internal device is 100B words. If the writer writes more than this amount of data before the reader removes anything from the buffer, the writer is put in a wait state.

To prevent this the size may be increased by the system supervisor through the @SINTRAN-SERVICE-PROGRAM command *CHANGE-BUFFER-SIZE.

For an internal device the input buffer is used. The input and the output part have a common buffer, logically a part of the input datafield and the output datafield is not used.

Before the changed buffer size becomes effective, the system must be restarted ("warm start"). The modification may be made permanent, surviving a cold start, by modifying the save area as well as the image area.

Buffer sizes cannot be increased beyond the space available in the POF area. If the POF area is already overcrowded, the system supervisor may release some space by reducing the buffer size of the internal devices. This should not be done if the system is close to saturation with respect to CPU time, as it may lead to more frequent process switching (due to full buffers). In a heavily loaded system, even a small increase in overhead may be significant.

12.3 Word oriented internal devices

Each INBT or OUTBT call on a byte oriented internal device transfers 8 bits of data. This may be modified to 16 bits or one ND-100 word and the device is then termed word oriented. Each device can be modified individually.

Modification must be made by the system supervisor after Sintran has been loaded. The patch is described in the MODIFICATIONS memo delivered with the Sintran diskettes and can be made permanent by patching the save area or temporary until the next cold start by patching the memory image.

Modifying the device to word operation reduces the number of I/O monitor calls, thereby reducing system overhead somewhat compared to one byte INBT/OUTBT calls. The same savings in overhead can usually be obtained by using the multibyte monitor calls B4INW, B8INB, M8INB, M8OUT and B8OUT. More important, a word structure may more closely reflect the structure of the data transmitted.

If the ISIZE call is used on a word oriented device, the number of words in the input buffer is returned. Thus it represents the number of (16 bit) INBT calls that can be executed before the buffer is empty.

In other respects a word oriented device is operated in the same way as a byte oriented one.

12.4 Block oriented internal devices

Block oriented devices are primarily used where large amounts of data must be moved rapidly from one segment to another.

Like byte oriented internal devices, they use buffer area owned by the operating system. The buffer is not a part of the data field, but allocated dynamically in units of one page at a time. When moving large amounts of data, the overhead is greatly reduced compared to byte oriented devices. Each read or write operation may transfer up to a full page (2048 bytes) of data.

If block oriented devices are going to be used, this must be specified when ordering Sintran.

12.4.1 Device numbers of block oriented devices

Due to the buffer area required, a maximum of five block oriented devices can be ordered with a Sintran system. The device numbers run from 200B to 204B, overlapping the range used for byte oriented devices. In a system delivered with both byte and block oriented devices, the block oriented ones are those with the lower numbers (unless otherwise requested).

12.4.2 Reserving a block oriented internal device

Reservation of a block oriented device the same as for a byte oriented one:

```
CALL RESRV (200B, INPUT, WAIT)
```

12.4.3 Reading and writing

The RFILE, WFILE and MAGTP calls may be used on block oriented internal devices. However random access is not possible, the block number parameter of the RFILE and WFILE calls is ignored (treated as if -1 was specified, next block). MAGTP is more commonly used:

```
INTEGER WREAD
PARAMETER (RREC = 0)
CALL MAGTP(RREC, ARR, 202B, 400B, WREAD)
```

400B words (256 decimal) is read from internal device 202B by the Read Record function. The read data is stored in the array ARR and if less than 400B words are available the number actually read is returned in the WREAD variable.

A block of data may be written by a similar MAGTP call:

```
PARAMETER (WREC = 1)
CALL MAGTP(WREC, ARR, 202B, 400B, 0)
```

In the Write Record call, the last parameter is not used but a dummy parameter must be supplied as in the example above. The Read and Write Record functions are the only ones permitted on internal devices.

Error status is set as for ordinary file access, e.g. access beyond the buffer size results in a "No such block" condition. The error code is returned in the A register or in Fortran in the system variable ERRCODE.

12.4.4 Clearing the buffer

The programmer may want to clear the buffer of a block oriented internal device, in a manner similar to the CIBUF call for byte devices. This can be done through the MAGTP function CLEAR-DEVICE:

```
PARAMETER (CLEARDEV = 21B)

CALL MAGTP(CLEARDEV,0,202B,0,0)
```

The second, fourth and fifth parameter are dummy parameters.

12.5 Sharing a segment

A segment may contain parts of or the complete code of several programs. To load several programs to one segment, all one needs to do is load several files in succession. If the programs are all in one file as a result of one compilation, that file is loaded.

A compiler usually generates a MAIN control byte in the BRF code for each PROGRAM statement or equivalent in other languages. (In Fortran the PROGRAM statement is optional, but is implicit if a compilation unit does not start with a SUBROUTINE or FUNCTION statement.) Each MAIN control byte loaded by the RT loader reserves one RT description.

These programs may keep data in a COMMON block, referred to by two or more of the programs. A modification of a common block location by one program then immediately affects other programs as well.

```
        PROGRAM A, 30
        CHARACTER * 80  AMSG
        COMMON /AB/AMSG
        AMSG = 'This goes from A to B'
        END

        PROGRAM B, 30
        CHARACTER * 80 BMSG
        COMMON /AB/BMSG
        CALL RESRV(52,1,0)
        WRITE(52,100) BMSG
100     FORMAT(1X,A80)
        END
```

After compilation both files containing the two programs are loaded by the NREENTRANT-LOAD command:

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NREENTRANT-LOAD A,,
NEW SEGMENT NO: 173
*NREENTRANT-LOAD B,,
*END-LOAD
*EXIT
```

The two programs are started from terminal 52 (A is started first so that the string assignment is done in A before it is printed in B):

```
@RT A
@RT B
@LOGOUT
 12.14.15        7 OCTOBER     1981
--EXIT--

This goes from A to B
```

The program need not be placed on the same segment as the data area. Each of the programs may use a separate segment for the program and read the common data block from a second segment, or one program may

be on the same segment as the common block while another one is on a separate segment. The use of several segments to separate the common areas is discussed in detail in chapter 14.

The data need not be declared in a common block, any address within the segment can be referred to. However, common blocks provide a convenient symbolic notation for referring to common data.

12.5.1 **Access conflicts**

As the two programs are active within the same machine, only one is executing at a time. Concurrent access to the same memory location are therefore impossible.

One program may be interrupted in the middle of the updating of a data structure. An example of this is shown in the previous chapter (section 11.3), where one solution is illustrated, using a semaphore to protect the common data. Use of such synchronizing mechanisms is the responsibility of the programmer.

12.6 Communication with background processes

Communication between RT and background programs is mainly used for a background system to request services from e.g. a database system running as a real time process. The results of such requests may also be returned to the background program.

Such communication may go through internal devices, files or the XMSG communication system. XMSG is described briefly in chapter 20.

12.6.1 Internal devices

The RESRV call is available in background as well as RT programs and may be used to reserve the input or output part of byte, word or block oriented devices. The ordinary WRITE and READ Fortran statements may be used on byte devices, MAGTP or RFILE/WFILE on block devices.

If access is attempted that causes an IOWAIT condition (read when buffer empty or write when buffer full), the background program enters a waiting state. This may be interrupted by pushing the 'escape' key on the terminal (provided escape function has not been disabled). This causes the program to be interrupted. The internal device will not be released until the user logs out or releases it through a monitor call.

If NOWAIT mode is specified for an internal input device in background mode and there is no input available, an end-of-file condition is returned. NOWAIT specified for an internal output device has no effect, control is not returned to the program before the transfer is completed.

12.6.2 Using permanent files

As described in detail in the next chapter, background and RT programs use common files. A file written by an RT program may be read by a background program and vice versa, as long as the writing program has formatted the data in the file so that the reading program can understand it.

Using files for communication has one significant drawback, namely speed. File operations are generally slower than other communication methods, due to the file system overhead. Also, as long as one program writes to the file, no other program can read it.

When the speed is of less importance, files are suitable for transferring large amounts of data. The cost in terms of system overhead per byte is small and synchronization between writer and reader is unnecessary. Data in a file may be read several times by different users, occupying no resources other than disk space.

12.6.3 Internal devices as "peripheral files"

Sometimes file access monitor calls are more convenient than internal devices for communicating data. In languages that do not have a library of RT monitor call routines, this is the only possible solution if the programmer does not have experience in writing assembler routines.

(A library is usually available for those languages used in writing the RT program, but the background program with which it communicates may be written in any other language. In most cases it is also written by less experienced programmers.)

User SYSTEM can define names for any device, including internal ones, using the Sintran command @SET-PERIPHERAL-FILE. This command associates a file name with a device number and the device may be reserved by a background program by opening the file, in exactly the same way as any other file is opened. The type of the file is blank (none).

If the file is opened for sequential read, the input part of the device is reserved. If it is opened for sequential write, the output part is reserved. The file system returns a device (file) number from the OPEN call. Access to the file may be through this file number or through the static device number (in the range 200B-237B).

The internal device can also be accessed through its peripheral file name from RT programs. This activates the file system, significantly increasing overhead compared to reserving the device through RESRV.

The reason for using the file name may be that a language where the RESRV monitor call is unavailable is used even for the RT part or that the device should be available to a number of programs.

Any of a number of programs may read from a file, any program may write to it. This is described in more detail in the next chapter. For an internal device this means that the first program to perform an input call reads and removes the first byte or bytes from the input buffer, programs performing a read at a later time do not see those bytes already read.

12.7 Survey of communication methods

Below is a summary of essential properies of various communication methods

SPEED time consumed is proportional to message length unless noted

CAPACITY how suitable the method is for transferring large amount of data. Low: message length < 250 bytes, Moderate: mesage length up to 2 K bytes, High: message length: several K bytes

MULTIMACHINE
indicates whether sender and reciever may be in different CPUs

MULTIPLE READERS
concurrently, without releasing and recieving

MULTIPLE WRITERS
concurrently, without releasing and recieving

BACKGROUND ACCESS
whether it can be used for communication with background programs

SYSTEM OVERHEAD
CPU work performed by the monitor

PROGRAM OVERHEAD
preparations required before a message is transmitted

IDENTIFICATION
how the programmer identifies the receiver

MAY REPLACE
fully or partly, not necessarily always

RTCOMMON:

Speed:	Immediate
Capacity:	System generation parameter, usually moderate
Multimachine:	No
Multiple readers:	Yes
Multiple writers:	Yes
Background access:	No
System overhead:	No
Program overhead:	No
Identification:	Memory address/symbolic COMMON block name
May replace:	Shared segment, internal devices, semaphores XMSG (single CPU)
Remarks:	Never swapped

Shared segment:

Speed:	Immediate (unless segment is swapped out)
Capacity:	Moderate/large
Multimachine:	No
Multiple readers:	Yes
Multiple writers:	Yes
Background access:	No
System overhead:	No
Program overhead:	No
Identification:	Memory address/symbolic COMMON block name
May replace:	RTCOMMON, internal devices, semaphores XMSG (single CPU)
Remarks:	User program is responsible for all protection of area

Semaphore:

Speed:	High
Capacity:	No
Multimachine:	No
Multiple readers:	Not applicable
Multiple writers:	Not applicable
Background access:	Yes
System overhead:	Moderate
Program overhead:	Moderate
Identification:	Device number
May replace:	Logical RTCOMMON variables
Remarks:	

Internal device, byte oriented:

Speed:	High
Capacity:	Low
Multimachine:	No
Multiple readers:	No
Multiple writers:	No
Background access:	Yes
System overhead:	Moderate
Program overhead:	Moderate
Identification:	Device number
May replace:	RTCOMMON, internal block devices XMSG (single CPU)
Remarks:	

Internal device, block oriented:

Speed:	High
Capacity:	Moderate/High
Multimachine:	No
Multiple readers:	No
Multiple writers:	No
Background access:	Yes
System overhead:	Low, in particular for large amounts of data
Program overhead:	Low
Identification:	Device number
May replace:	Internal byte devices, XMSG (single CPU)
Remarks:	Option

ND-net:

Speed:	Moderate/low
Capacity:	Moderate
Multimachine:	Yes
Multiple readers:	No
Multiple writers:	No
Background access:	Yes
System overhead:	High
Program overhead:	Moderate
Identification:	Configuration dependent channel number
May replace:	Multimachine XMSG, transfer via files
Remarks:	Multimachine communication only.

Files:

Speed:	Low
Capacity:	Very high
Multimachine:	Possible
Multiple readers:	Yes
Multiple writers:	No (usually)
Background access:	Yes
System overhead:	High
Program overhead:	Moderate
Identification:	Symbolic file name
May replace:	ND-net, internal block devices
Remarks:	Permanent data which may be read several times

XMSG:

Speed:	High
Capacity:	Moderate/high
Multimachine:	Yes
Multiple readers:	Yes
Multiple writers:	Yes
Background access:	Yes; also direct tasks and drivers
System overhead:	Moderate; less for large messages
Program overhead:	Moderate (Setup: Moderate/high)
Identification:	Symbolic port name
May replace:	Internal devices, ND-net, shared segment, RTCOMMON
Remarks:	Variable size messages, overhead does not increase with size of message. May be used as multi-CPU semaphore or

(single- or) multi-cpu critical region.
Indirect access (capability transfer) possible.

13 FILE ACCESS FROM RT PROGRAMS

RT programs access files in much the same way as background programs. But as there is generally no terminal connected to the program or a logged in user corresponding to the program, some details are handled differently.

The SINTRAN command @RTENTER must have been executed since system start before RT programs can use files. This is a one time operation normally found in the HENT-MODE file executed at every cold start. It is assumed below that @RTENTER has been executed, if not, user RT or SYSTEM can give the command at any time before the first program using files is started.

13.1 User and file name

The same files are accessed by RT and background programs, consequently the naming rules are the same. Any user's files can be read, provided the user name enclosed in parentheses preceeds the file name. Default user name is RT and the access rights to the file (read, write, directory) are determined by user RT's access. If the file is not found and no user name is specified, the files of user SYSTEM are searched.

13.2 The files accessible to a program

Each background user has his own list of open files and in general two users may not modify one file concurrently (or even one user modify while another one reads). However, Files opened by one RT program are available to all other RT programs. A set of programs operating on data in a file need not close the file before giving control to another program in the set. On the other hand, the file is not protected by access from other RT programs.

A semaphore, RTCOMMON or other common variable may be used to control the use of a file within a group of programs. There is no guarantee that other programs obey such regulations.

13.3 The file number

When a file is opened, a device number is returned corresponding to the device number used for semaphores, internal and external devices.

There is a one to one correspondence between a semaphore/device and a device number; the association between a file and a device number is dynamically determined when the file is opened. The first file opened, by any RT program, receives the device number 100B, the second one 101B and so on.

When a file is closed, the device number is released. The same number can be reused when the next file is opened by an RT program. The probability that a file closed and reopened later receives the same device number is small.

The device numbers used for open files go from 100B and upwards. Unless SINTRAN is ordered with a larger file table, no more than 45 files can be opened by RT programs at one time. Each spooling program running keeps one file open, reducing the number of files available to user programs.

There is no equivalent to the terminal scratch file (sometimes referred to as "file 100") used as a storage area for temporary data by some background systems, but a file opened from an RT program may be defined as a temporary file through the SINTRAN command @SET-TEMPORARY-FILE.

13.4 File numbers of peripheral files

If a file is a peripheral file, defined by the SINTRAN command @SET-PERIPHERAL-FILE or the SPEFI call (MON 234) the file number returned is that of the device and is outside the range 100B:137B.

The device number can be used exactly like a file number. A device should not be accessed through the file system (OPEN/CLOSE) and as a device (RESRV/RELES) at the same time. If opened through the file system it should be closed through the file system and should not be released through a RELES call.

13.5 The Fortran file number

By convention, a file in a Fortran program is identified through a static (constant) integer less than 100. In an ordinary background program, the Fortran file number is translated via a table (local to the program) to a device number used by SINTRAN.

File handling is different in nonreentrant and reentrant Fortran. This also applies to the closing of files - see section 13.6

13.5.1 Nonreentrant Fortran

A connect table is used to hide the SINTRAN file number from the programmer. A program written in standard Fortran, running correctly as a background program, will run in real time if compiled in nonreentrant mode.

Due to the translation mechanism, there is no way to share files between programs. If the Fortran file number used in one program is transmitted to another program, there is no way the latter file table can be updated except by opening the file in the second program.

13.5.2 Reentrant Fortran

Transferring the identity of an open file to another program is often desirable. This is possible in Fortran programs compiled in reentrant mode (compiler command $REENTRANT-MODE).

The file number should be an integer variable. After the OPEN statement is executed, the variable contains the device number assigned to the file and when the variable is used later in input/output statements, the correct device number is used.

```
INTEGER FILENO
OPEN(FILENO, FILE='DATA', ACCESS='SEQUENTIAL')
WRITE(FILENO, 300) VAR1, VAR2, VAR3
```

An integer variable can be used as a file number in background programs, thus RT and background programs are source code compatible. In a background program, the file number variable should be in the range 1 to 99 when the OPEN statement is executed and it is not modified. When executed in a reentrant RT program, the initial value is ignored and the value is modified. The following is therefore legal in background as well as RT:

```
INTEGER IOFILE
IOFILE = 11
OPEN(IOFILE, FILE='NUMBERS', ACCESS='RW')
WRITE(IOFILE, 200) I1, I2
```

The file number variable should not be modified by the program after the OPEN statement has been executed!

Most other high level languages use a "file type" variable identifying the file in OPEN and I/O statements and the user need not be concerned about the device number at all.

13.6 Opening the file

A file is opened by the Fortran OPEN statement as shown above or by a statement or call in any language that is compiled to the MON OPEN call (MON 50). The relevant language manual should be consulted for parameters and other details. There are no limitations to the access method specified, sequential, random or direct transfer read or write - except those enforced in all file access (e.g. direct transfer allowed on contiguous files only).

A file can also be opened by a command:

```
@RTOPEN-FILE <file name> <access mode>
```

that is an equivalent to the @OPEN-FILE command, but the file is accessible to all RT programs. The default user name is RT, even if the command is executed by user SYSTEM. Default type is :SYMB. The file opened through @RTOPEN-FILE is not closed when the terminal from which the command was given is logged out.

The program(s) must be able to find the file number returned from the call to access the file. E.g. one RT program can reserve the terminal and prompt for the file number.

User RT or SYSTEM can list the files open by any RT program or through the @RTOPEN-FILE command by

```
@LIST-RTOPENED-FILES
OUTPUT FILE: TERMINAL

FILE NUMBER 000100 : (PACK-ONE:RT)S310:BPUN;1
FILE NUMBER 000101 : (PACK-TWO:DDS-SYS)NORDDIS:LOGG;1
@
```

When a file is opened by a command, the file number can be forced to a specific value by using the @RTCONNECT-FILE rather than @RTOPEN-FILE. The file number specified must be unoccupied and in the range 100B:121B, specified as an octal number not followed by B.

The numbers already occupied are found by the @LIST-RTOPENED-FILES command. If the file number is in use, an error message is returned:

```
@RTCONNECT-FILE
FILE NAME: OUT:SYMB
FILE NUMBER: 100
ACCESS MODE (R,W,RW,RX,WX,WA,RC,WC): RX

FILE NUMBER ALREADY USED
@
```

13.7 Closing the file

A file is closed by CLOSE (MON 43). This call is available in all high level languages as a complement to OPEN. On assembler level, the T register should contain the file number.

A file opened by an RT program is not closed when that program terminates. Any program can close a file opened by any other program. A file can also be closed by a command:

```
@RTCLOSE-FILE
FILE NUMBER: 100
@
```

This command should not be executed until all programs have terminated operation on the file. If a file number of -1 or -2 is specified (in the command or monitor call), all files open for RT programs are closed.

13.8 Closing of files on program termination

Program termination does not usually close any files. In **nonreentrant** Fortran, because the identity of the file cannot be transferred to another program, the file is closed when the program terminates normally. This ensures that the last block is properly written to disk, the buffer area is cleaned up and the file is closed.

If the program is prematurely aborted or terminated outside the control of the Fortran run time system (e.g. in an assembler routine), files are not closed. The buffers are not properly written back to disk (unless this is taken care of by the assembler routine). Thus, data in buffered files may be inconsistent.

In **reentrant** Fortran ($REENTRANT-MODE compiler command), the file can be accessed by other programs after the program opening it has terminated. When operations on the file are complete, the file must be explicitly closed by a call to the CLOSE routine; there is no automatic closing of the file.

Other high level languages vary with respect to closing files. File closing must be specified in a Basic program. Pascal automatically closes buffered files opened by the program. In MAC, NPL and Planc all files must be closed explicitly.

13.9 Reading and writing

Most monitor calls for file access are legal from RT programs. The exceptions are the RDISK (MON 5) and WDISK (MON 6) calls for writing to the terminal's scratch file and the MSG (MON 32) and IOUT (MON 35) calls for writing a string or an integer, respectively, to the terminal permitted.

Programmers using a high level language such as Fortran, Pascal, Planc etc., do not see these calls as such. They may assume that the code generated uses monitor calls permitted in foreground as well as background, for random and sequential access.

13.10 Block I/O

The primary calls for disk file access are RFILE and WFILE (MON 117 and MON 120). These are used exactly as in background and are available as Fortran routines:

```
CALL RFILE(FILENO,WAITFLAG,ARRAY,BLOCKNO,WORDS)
```

The WAITFLAG may be different from 0 in an RT program, which initiates the transfer and then returns control to the calling program. If the wait flag is zero, the calling program is suspended by putting it in an IOWAIT state until the transfer is complete.

Only one RT program at a time may execute file I/O at a time; several requests occuring faster than the file system can complete them are queued.

13.10.1 Checking the status of the transfer through WAITF

An RFILE/WFILE transfer request with a non-zero WAITFLAG causes the disk access to run in parallell with the program.

At any later time the status of the operation can be checked: a WAITF call can return a status (finished, not finished) and let the executing program continue or it can hold the program until the transfer is finished.

WAITF is available as a Fortran function:

```
      PARAMETER (WAITCOMPL = 0, IMMRETURN = 1)
      INTEGER STATUS, FILENO, WAITF
      EXTERNAL WAITF

C     Start an I/O transfer and go on computing

      STATUS = WAITF(FILENO, WAITCOMPL)

C     Perform calculations while waiting for IO to complete
```

The first parameter, FILENO, is returned from an OPEN call. The second parameter is 0 if the calling program should be held until the most recently RFILE or WFILE call operating on the file with the specified number has completed. It is 1 if the program should continue even if the transfer is still going on.

The STATUS function value is -1 if the transfer is not yet complete. If the function value is 0 it is complete, if positive, the value is a SINTRAN file system error code. An error code refers to incomplete execution of the WAITF call, not to the RFILE/WFILE call.

In assembler code, the A register points to the parameter list. The status value is returned in the A register:

```
          WAITF=121
          IMMRET=1

          LDA (PAR
          MON WAITF
          STA STATUS

PAR,      FILNO
          (IMMRET      % Return immediately
FILNO,    0            % Filled in when file opened
STATUS,   0            % -1: incomplete, 0: complete, >0: error
```

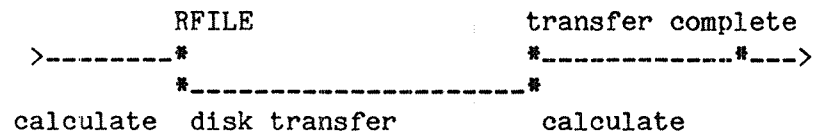
If a WAITF call specifies that the program should be held until transfer is complete, the return value is never -1.

13.10.2 Double buffering

The WAITF function used in conjunction with a nonzero wait flag in the RFILE/WFILE call can be used to implement double buffering. While one set of data is operated on, the next set is read in from disk or the previous one written back to disk.

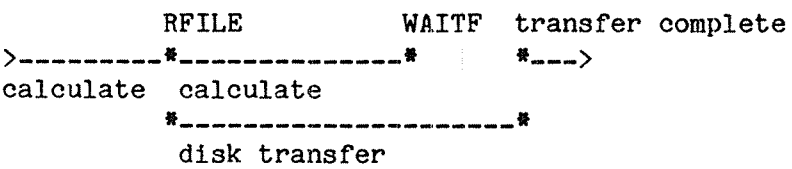
This utilizes the time spent waiting for the disk, in cases where it is possible to start a transfer some time before the data is needed. Equally important in a heavily loaded system, no other program is automatically started, reducing the administrative work of the RT Monitor. (In normal mode with the wait flag equal to zero, the calling program is immediately put in an IOWAIT state and the execution queue searched for another program to be started.)

WAIT FLAG = 0:

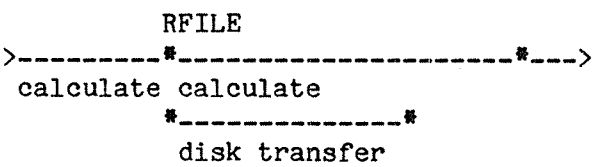


WAIT FLAG = 1:

(disk transfer time > calculation time)



(disk transfer time < calculation time)



In the theoretical "best case", the CPU operations are completed at the exact moment when the disk transfer is finished. This would give close to double the speed (50% reduction in elapsed time). In practice, the time saved is far less, typically 10-15%. This is due to the starting of the transfer and CPU time spent handling interrupts from other sources. If the priority of the program is low, some other program may use the CPU during the disk transfer.

Even though the savings are less than the theoretical best, they are often worthwile.

13.11 Character I/O

For sequential, unbuffered I/O, the INBT (MON 1) and OUTBT (MON 2) calls are used by the libraries. If large amounts of data are to be transferred, the overhead can be reduced by use of the "block" calls M8INB (MON 21), M8OUT (MON 22), B8INB (MON 23) and B8OUT (MON 24).

A string can also be input from a terminal by INSTR (MON 161), output through OUTST (MON 162). These two monitor calls are options.

13.11.1 **NOWAIT mode**

The program can continue execution while transfer to or from a character device such as a terminal is taking place by specifying NOWAIT mode for that device. This allows arbitrary time delays between the INBT/OUTBT call and the completion of the transfer; e.g. input can be requested from a terminal, and while the operator scratches his head the program can prepare the next output or access another device.

Use of NOWAIT closely resembles setting the WAITFLAG to non-zero in a RFILE or WFILE call to read a disk file.

NOWAIT is individually specified for each device and can be set and reset at the programmers discretion.

13.11.2 **Setting and resetting NOWAIT**

The NOWT call, MON 36, is used to set and reset the NOWAIT mode. The argument list consists of the device number or file number from an OPEN call, an I/O flag where 0 indicates input and 1 indicates output, and an on/off switch, 0: NOWAIT on, ><0: NOWAIT off.

```
        NOWT=36
        OPEN=50
        QERMS=65
        INP=0
        ON=0
        SEQR=1

        LDX (FILNAM
        LDA (SYMB
        SAT SEQR
        MON OPEN
        MON QERMS
        STA FILNUM
        LDA (NWPAR
        MON NOWT

FILNAM, 'TAPE-READER'
SYMB,   'SYMB'
FILNUM, 0
NWPAR,  FILNUM
        (INP
        (ON
```

Subsequent read operations from TAPE-READER:SYMB never cause an IOWAIT condition. However, the NOWAIT mode applies to the calling program only and only until the file is closed or NOWT is called with the third parameter different from zero.

13.11.3 Using the NOWAIT mode

Rather than putting the calling program into IOWAIT mode waiting for the transfer to finish, an I/O call immediately returns the status code END OF FILE (error code 3). The program can continue and operate on other data. After finishing the operations that can be performed independent of the I/O transfer, the program may execute an RTWT or HOLD call.

When the transfer is finished, a "break condition" occurs and the program is restarted similarly to an RT call. If the program is active, the repeat bit in the RT description is set.

13.12 The device buffer

Character devices opened for sequential read or write are accessed through a buffer much like that used for internal devices. For input devices the input source can be treated like a "writing program", for output the device is like a "reading program".

Two way devices have independent buffers for read and write. The size of these buffers may be modified in the same way as buffers for internal devices.

Special problems occur with input devices if the input is unpredictable:

The device writing to the buffer, e.g. a terminal, cannot be stopped in the same way as a program without affecting the input. When the buffer is full and input continues arriving, the characters are lost. Input is not honored until buffer space is made available by reading from the device buffer.

The current amount of data can be read by the ISIZE and OSIZE monitor calls. ISIZE returns the number of bytes in the buffer of the input part of the device, while OSIZE returns the amount of data that can be written to the output part without overflowing the buffer.

The calls described below are available in background as well as RT programs.

13.12.1 The amount of data in the input buffer

The ISIZE call is available as an integer Fortran function:

```
UNREAD = ISIZE(52)
```

This call returns the number of bytes in the ring buffer of the terminal with device number 52, i.e. the number of characters that can be read without going into an IOWAIT state. The device need not be reserved before the ISIZE call is executed.

The ISIZE (MON 66) call expects the device number in the T register and returns the number of bytes in the A register:

```
        ISIZE=66

        SAT 64      % 64B = 52 decimal
        MON ISIZE
        STA COUNT

COUNT,  0
```

13.12.2 The amount of data in the output buffer

OSIZE, MON 67, returns the number of bytes that can be written to the device without causing an IOWAIT. The number of bytes not yet output on the device can be found by subtracting the value returned from OSIZE from the size of the buffer.

OSIZE is applicable only to devices with an output datafield. Internal devices use the data buffer in the input field, see section 12.2.5

13.12.3 Clearing the input buffer

If the program reading from a device wishes to reset the number of bytes buffered to zero, it can do so by the CIBUF (MON 13) call.

The CIBUF call is functionally equivalent to reading and immediately discarding all input that is available in the buffer at the time of the call. This is legal only if the input part of the device has been reserved.

In assembler, the T register must contain the device number. After return the A register contains an error code if any. In case of error, return is to the first location following the MON instruction, while ordinary return is to the second instruction following the call ("skip return").

13.12.4 Clearing the output buffer

The COBUF call cancels any output that has been written to a device but not yet read by the recipient and is legal only if the output part of the device has been reserved. In addition to clearing the buffer, a "break condition" is generated on the input side of the device. This causes the program reserving the input part to be activated if it was in an RTWT or HOLD state.

COBUF (MON 14) is available in Fortran; the Fortran and MAC call sequences are exactly as for CIBUF above.

13.13 Optimizing file access

RT programs are often highly time critical and dependent on predictable response times. Many applications spend the most of their time waiting for some kind of I/O and performance can be greatly improved by selecting an appropriate I/O strategy. This includes both reduction in the response time of the program and in the total system load.

Often this requires modifications to the program, even when a high level language is used. However, in most cases the program still conforms to the standard definition of the language or the compiler indicates deviations and extensions to the standard.

13.13.1 The Fortran access mode

The Fortran OPEN statement contains a parameter ACCESS=<acc>, where <acc> by Norsk Data conventions is either R, RW, W for sequential access, RX, WX for random access. This gives unbuffered I/O: each READ or WRITE statement causes a disk access.

In nonreentrant Fortran, files used for sequential access may be buffered. When the first data element is read, an entire disk page (2048 bytes) is read to a buffer. The buffer is allocated in the data area of the user program.

The next time an input operation is requested, the data is found in the buffer, preventing the disk access. Also, the program does not enter IOWAIT state. No operating system request is required until the buffer is exhausted, at which time another page is read. The number of monitor calls (and in the case of I/O process switches in the system) to perform sequential I/O is reduced by a factor of up to 2048 compared to unbuffered INBT calls.

A similar mechanism is used for output. A data value is not written to disk immediately, but is put into a buffer. When the buffer is full, the entire buffer is output with one call.

Buffered I/O is selected by specifying access codes

READ	- buffered input
WRITE	- buffered output
SEQUENTIAL	- buffered input and/or output
DIRECT	- buffered input and/or output

The ANSI-77 standard requires SEQUENTIAL to be used. However, the file is then opened with both read and write permitted (unless the access permission associated with the file restricts it to read only). For security reasons, this may be undesirable. Using the READ and WRITE codes also reflects the intended use of the file, but is an Norsk Data extension.

The size of the area used for I/O buffers limits the number of files that are concurrently used for buffered I/O. The user specifies the size of the buffer by the RT loader command *SET-IO-BUFFERS:

```
*SET-IO-BUFFERS
NO. OF 1K BUFFERS: 6
```

The number of 1K buffers is the maximum number of files that are buffered. If more files are open concurrently for sequential access (in the example above 7 or more), the last opened files are not buffered even if the access code was specified as READ, WRITE or SEQUENTIAL. By default, no buffers are allocated. *SET-IO-BUFFERS apply to FTN only. Other languages, including the ANSI-77 standard FORTRAN-100, use file buffers defined in the run time libraries.

Random access files are never automatically buffered. In most cases, the next data item to be read is not the next in the file or even within the same disk page, thus nothing is gained. Random access is compiled to a RFILE/WFILE call, reading an entire logical record in one monitor call, thus the monitor call overhead is moderate compared to sequential byte-by-byte access.

Sequential files may be accessed fairly efficiently even un-buffered if the MAGTP monitor call is used. In spite of its name, this call may be used on any sequential file, reading a record of a specified length. The call is available in Fortran:

```
ISTAT = MAGTP(IFUNC, IARRAY, IUNIT, IPAR1, IPAR2)
```

IFUNC may take a value from 0 to 46B; some functions are relevant to specific devices only. The most useful are 0: Read Record, and 1: Write Record. IARRAY is the data block to be transferred, and IUNIT is the SINTRAN open file number. IPAR1/IPAR2 are dependent on the function code; for Read Record IPAR1 is the programmer specified maximum number of words to be read, IPAR2 the returned as the actual number of words transferred. For Write Record IPAR1 specifies the number of words to be written, IPAR2 is dummy.

For information on other MAGTP function codes, see SINTRAN III Reference Manual.

13.13.2 Contiguous files

Files created by enclosing the file name in double quotes ("FILENAME") are indexed files. Initially, an index page is allocated that contains no user data but the addresses of the data pages. These can be scattered all over the disk and when a new page is needed, it can be allocated anywhere on the disk as long as its address is entered into the index page. File expansion causes no problems.

To save looking up the index table, a contiguous area of the disk may be reserved for the file. This allows the file system to calculate the address of the referenced record from the record number and the file address, saving the reading of an index page. The increase in speed is significant, in particular if the file system is heavily loaded, and no modification needs to be made to the program using the file.

The disadvantage of contiguous files is that the maximum size of the file must be known when the file is created. If it grows beyond this size, a new and larger file must be created and contents of the old file copied.

If the logical file addresses used are disjunct ("holes" in the file), this space must still be allocated to the file. "Holes" are not permitted in a contiguous file; it must cover the entire address range from 0 up to the highest logical address used.

A contiguous file is created by the SINTRAN command @CREATE-FILE or MON CRALF (MON 221), specifying the number of pages required.

13.13.3 **Direct transfer**

Disk-to-memory transfer usually goes to a buffer area maintained by the operating system, not a part of the user address area. From this buffer, a memory-to-memory copy to the user area is executed. The rationale behind this is that if the disk transfer went directly to the user area, a page fault might occur during the transfer and this could only be handled with great difficulty, the WIP bit would have to be set explicitly, and several transfers might be necessary if the block did not coincide with page boundaries.

Secondly, some disk units are not capable of transferring an arbitrary number of bytes, but is limited to an integral number of disk sectors. The copying from the system to the user area allows an arbitrary number of bytes to be copied, fitted to the user defined logical block size.

Direct transfer from disk to user area is possible, but with some limitations on the flexibily:

- the area affected by the transfer must be FIXCed in a contiguous area of memory to prevent page faults during the transfer
- the block size must be a multiple of the disk sector size
- the file must be contiguous
- the file must be read or written by the monitor calls RFILE/WFILE only (WAITF is permitted)

Where these limitations are acceptable, direct transfer is the most efficient file access. In particular where large blocks of data (up to one disk cylinder) are transferred in each call, the speed of the transfer may approach the hardware speed of the disk (i.e. two to ten times faster than normal transfer).

If DIRECT (random access), SEQUENTIAL, SPECIAL, D, READ or WRITE (sequential access) is specified in the OPEN statement, the Fortran library attempts direct transfer if all the conditions listed above are satisfied. Otherwise, an ordinary file open is performed.

13.13.4 Absolute transfer

A program running on PT0, in ring 2 and fixed in contiguous memory (FIXC monitor call) can perform low level disk transfer through the ABSTR (MON 131) call. This is the basic monitor call used by the file system itself. This call bypasses the file system and the area to be transferred is specified by physical memory and disk addresses.

Compared to direct transfer, the gain in speed is insignificant. However, ABSTR can be used to access disk pages that cannot be reached through the file system, e.g. the index pages of files. This require a detailled knowledge of the file system and SINTRAN and is configuration dependent.

The ABSTR call must be used with great care; incorrect use may corrupt the disk including the operating system and user files.

13.13.5 Reading and writing disk pages

SINTRAN H and later versions include monitor calls for reading and writing arbitrary disk pages without the programmer being concerned about physical differences between various disk units.

The directory must be reserved through the REDIR monitor call before the reading or writing takes place.

The example below will read an entire floppy disk, 148 pages, into segments 300, 301 and 302. Segment 300 and 301 each holds 64 pages, 302 the last 20. The alternative page table mechanism is used, and the data segment exchanged between each transfer.

The disk pages are read in the physical order on the disk, and the RT program is expected to interpret the file system tables in order to access data pages in logical sequence. Performing these operations on data segments rather than on the floppy disk is an order of magnitude faster.

The A register should contain the address of a double word containing the disk page address, X contains the memory buffer address, T the directory index and D the number of pages to be transferred.

```
         RDPAG=267
         REDIR=246
         RLDIR=247
         ALTON=33
         ALTOF=34
         FLOPP=5        % Directory index of floppy disk
                        % (configuration dependent)
         SAT FLOPP
         MON REDIR      % Floppy must be reserved
         JMP ERR        % Error exit: A = error code
         SAX 0          % X = Memory address for transfer
         SAA 100
         COPY SA DD     % D = Number of pages
         LDT (P0
         LDA (PT3       % Use T temporarily for page no addr
         MON ALTON      % Activate two bank mechanism
         COPY ST DA     % A = address of double word 0
         SAT FLOPP      % T = Directory index
         MON RDPAG      % Read pages 0-77B into seg 300
         JMP ERR        % Error return: A = error code
         MON ALTOF
         LDT (PAR1
         MON MCALL      % Replace data segment with seg 301
PART2,   SAX 0          % X = Memory address for transfer
         SAA 100
         COPY SA DD     % D = Number of pages
         LDT (P100
         LDA (PT3
         MON ALTON      % Reset to two bank mode
         COPY SY DA     % A = first disk page, 100B
         SAT FLOPP
         MON RDPAG      % Read pages 100B-177B into seg 301
         JMP ERR
         MON ALTOF
         LDT (PAR2
         MON MCALL      % Fetch data segment 302
PART3,   SAX 0
         SAA 24         % 24B = 20 decimal pages
         COPY SA DD
         LDT (P200
         LDA (PT3
         MON ALTON      % Reset to two bank mode
         COPY ST DA
         SAT FLOPP      % T = Directory index
         MON RDPAG      % Read last 20 pages
         JMP ERR        % Error return: A = error code
         MON RLDIR      % Release floppy for other use
         JMP ERR        % Error return: A = error code
          . . .

PT3,     (3
PAR1,    PART2
         (177701        % Replace right hand segment with 301
PAR2,    PART3
         (177702        % Replace right hand segment with 302
P0,      0; 0           % Double word disk page 0
P100,    0; 100         % Double word disk page 100
P200,    0; 200         % Double word disk page 200
```

The WDPAG call, MON 270, will write rather than read the specified pages. The parameters are as for RDPAG.

13.13.6 Priority while performing file operations

When one program performs file operations, other programs must be prevented from accessing the same data, causing inconsistencies in the data structure. On the lowest level this is controlled by semaphores internal to the operating system.

A low priority process may obtain a semaphore locking e.g. an entire directory. Normally, such a lock is kept for very short periods of time; it is however possible that a higher priority process enters the system and starts executing. The semaphore may then be reserved for a long period, prohibiting any other access to the directory.

Programs performing file I/O should be given sufficiently high priority while they execute file system calls to ensure that they are not unneccessarily waiting for CPU resources to complete the call. Even if the priority is raised before each I/O-call and reduced immediately after, the saving of process switches easily outweighs the extra overhead when block I/O is used.

If the program can read or write data in larger chunks, this will reduce the number of access conflicts to the file system semaphores, and will also utilize the internal file system buffering and most disk units better.

13.14 Special monitor calls for input and output

The LASTC and EXIOX calls are used for special purpose I/O; for all ordinary applications, other calls are employed. This section can be skipped by users who communicate with devices in the standard way.

Other special monitor calls are available for control of special I/O equipment and communication purposes. These calls are described in the Nord Process I/O Software Guide (ND-60.093) and the SINTRAN III Communication Guide (ND-60.134).

13.14.1 Read the last character input from a device

Terminal input is normally read by reserving the appropriate logical device and executing the INBT call (directly, or indirectly by use of high level language READ routines). Sometimes it is inconvenient to reserve the terminal.

LASTC gets the last (most recently) character typed on a terminal; the terminal need not be reserved.

In MAC, the parameter list pointed to by the A register contains the logical device number of the terminal. This is consistent with other RT monitor calls, but different from MON INBT. Normal return is to the second location following the MON instruction ("skip return"), error return to the first, compatible with the MON INBT call. The character read is returned in the A register.

LASTC is avaiable in the Fortran library as an integer function, returning the ASCII value of the last character typed, with parity. To read the last character typed at terminal 52 in a Fortran program, execute the call

```
KAR = LASTC(52)
```

Or in MAC:

```
        LASTC=26;  QERMS=65

        LDA (TERM
        MON LASTC
        MON QERMS     % Print error message (see ch. 16)
        STA KAR

TERM,   (64           % 64B = 52 decimal
KAR,    0             % location where the last typed
                      % character is stored.
```

LASTC does not replace INBT for these reasons:

- the character returned is always the last one. If a terminal operator types a new character before the first one is read, the first one is lost

- there is no way to detect that a new character has been entered; the return is always immediate. If no new character has been entered, the same character is returned a second time. The only way to detect that a new character has been typed is to compare with the previous character read.

- there is no way to distinguish one character entered twice in sequence from a single character input

LASTC is valid for internal devices and returns the byte last written to the device. The byte is not removed from the ring buffer.

13.14.2 Echo and break modes of a terminal

An RT program can set the echo and break modes of a terminal with which it communicates through the ECHOM (MON 3) and BRKM (MON 4) calls, possibly differently for each terminal if more than one is used. The device number is specified in the T register or as the first parameter of the Fortran call.

To disable echo entirely and break on every character input from terminal 52, the MAC call is

```
TER52=64
ECHOM=3
BRKM=4
NOECHO=-1
ALWAYS=0

SAT TER52
SAA NOECHO
MON ECHOM
SAA ALWAYS
MON BRKM
```

13.14.3 Disabling and enabling the user break function

Special purpose RT programs may read the ESC (escape, ASCII code 33B) character as ordinary input. It may also be necessary to suppress the special handling SINTRAN gives the ESC. E.g. even though a terminal can be released from a background program through PRLS, as soon as the ESC key is depressed, the background program attempts to reserve the terminal.

The ESC character is handled as ordinary input if the DESCF (MON 71) call has been executed:

```
DESCF=71
TER52=64

SAT TER52
MON DESCF
```

Special handling of the ESC character can be reinstated through EESCF (MON 72). Both these calls are available in background programs as well, but the device number in the T register is ignored and the call applies to the current terminal.

If the RT program should trap the ESC character in order to perform cleanup before the program is terminated, and then let the user log in as an ordinary background user, the RT program must after reading the ESC and releasing the terminal execute an RT BAKnn call to start the login-sequence. (BAKnn is the name of the background program of the terminal.)

The user break (escape) character may be redefined by the MSDAE call (MON 227), which also sets the disconnect character for remote connection. The A register contains the user break character in the least significant byte, the disconnect character in the most significant byte. To change user break to ASCII NUL (control @) for terminal 52, but keep the disconnect character DEL (177B, standard value), execute:

```
MSDAE=227
TER52=64

SAT TER52
LDA (177400      % DEL (with parity) and NUL
MON MSDAE
```

The parity of both characters are significant; if specified with odd parity, no break or disconnect is executed. User SYSTEM may also define the user break character through the command

```
@DEFINE-ESCAPE-CHARACTER <term no.> <ASCII value>
```

The ASCII value is specified as an octal number. The current user break and disconnect characters may be read by MGDAE, MON 230:

```
MGDAE=230
TER52=64

SAT TER64
MON MGDAE
SHD SHR 10      % Shift out user break char
STA DISCH       % Store disconnect char
SAA 0
SHD 10          % Shift in user break char
STA UBCHR       % Store user break char
```

MSDAE and MGDAE are available in SINTRAN version H and later.

13.14.4 Executing an IOX instruction

The IOX instruction is the basic I/O instruction used by the operating system to write data or control signals to the interface registers of various external devices. Using this instruction properly requires knowledge of the system beyond the scope of this manual.

The IOX machine instruction is privileged and can be executed in ring 2 and 3 only. If an RT program in ring 1 or 0 wishes to execute an IOX, it can do so through MON EXIOX (MON 31). The argument list pointed to by the A register contains the value the A register should have when the IOX is executed and the device register address (corresponding to bits 0:12B of the IOX instruction). After the monitor call, the contents of the A register are the same as if the instruction was executed directly.

The device register address must be present in the operating system IOX table. The system supervisor can prohibit access to devices by removing them from this table through the @SINTRAN-SERVICE-PROGRAM command *REMOVE-FROM-IOX-TABLE.

MON EXIOX is available in Fortran as an integer function call:

```
        INTEGER EXIOX, AREGIN, AREGOUT, DEVREGADR

        AREGOUT = EXIOX(AREGIN, DEVREGADR)
```

In MAC:

```
        EXIOX=31

        LDA (PLIST
        MON EXIOX
        STA AREGOUT

PLIST,  AREGIN
        (164314   % Set the speed of terminal 1
AREGIN, (210      % Code 210 = 9600 baud
AREGOUT,0
```

An IOX instruction can also be executed by a command:

```
@EXECUTE-IOX <value> <device register address>
```

The parameters have the same meaning as for the monitor call and the result of the operation is written on the terminal. All input and output values are octal.

13.14.5 Set control information for a device

The IOSET (MON 141) call is used to explicitly load the control register of a peripheral device. This operation is normally performed by the I/O monitor call routine, but in some cases the user program may require certain functions not available through the ordinary calls.

The user should consult the documentation for the device in question, as the control codes are device dependent. A general discussion of device control is found in ND-06.016 NORD-100 Input/Output System.

The parameter list contains the device number, an I/O flag, the RT description address of the program reserving the device (0 = calling program) and a control code. An open file number cannot be specified as the device. If the device is not reserved by the specified program, the function is not executed, but gives an error return.

The control code -1 is common to most devices, indicating "Reset device". E.g. if nonalphabetic information has been sent to the line printer, logical device number 5, causing it to print a lot of rubbish, it can be stopped by the call

```
        PRINTER=5
        OUTPUT=1
        RESET=-1
        WHDEV=140
        IOSET=141
        QERMS=65

        LDA (PAR
        MON WHDEV           % Get the owner (reserver)
        JAZ HUH             % Nobody?????? Go to error routine
        STA OWNER           % Put the owning RTprogram
                            %  into parameter list
        LDA (PAR
        MON IOSET           % Clear device
        MON QERMS           % Error return
          .
          .

OWNER,  0
PAR,    (PRINTER
        (OUTPUT
        OWNER
        (RESET
```

14 MULTIPLE SEGMENT PROGRAMS

A program may be using one single segment, it may be using two or may be using a practically unlimited number of segments over a period of time, by replacing one of the currently active segments with another.

As applications grow, more and more of them will benefit significantly from being put on several segments. The justification for using several segments is very dependent on the application.

* The program has run out of space in the 64K addressing area. This is the main reason why the SINTRAN III system requires a condsiderable number of segments.

  If the application is modularized, routines used at one point in time are put in one segment. When these routines are no longer needed, the space they occupied in the addressing area is loaded with another segment with a different set of routines. This closely resembles the "overlaying" mechanism available in the Fortran library, but is significantly faster.

* Two segments may be placed on different page tables, 64 pages available for each, doubling the area available without any segment switching (the memory management mechanisms used for this are described in chapter 3).

* Two programs may want to share data. They may then be placed on the same segment (see section 12.5), but an error (e.g. addressing an array out of range) may destroy the "innocent" program. If the two programs are placed on separate segments, and a third segment used for data, one program cannot be destroyed by errors in the other.

* A program may be designed to work on several sets of data. The data segment may then be replaced with second, third, a fourth one... without any modification to the program segment.

* An application may be split into modules with different requirements. E.g. some routines may be time critical and require fixing in memory, while the major part of the routines are more or less independent of response time and should not occupy memory by being fixed. They may even be located on a demand segment.

* Different protection (ring or page) may be desired for different parts of the program or for the data and the instructions.

* Some system information may only be obtained by reading from one of the system included segments.

Segment switching is initiated through monitor calls. A call may replace one of the two segment numbers in the ACTSEG location in the RT description with another one, specified in the call.

There are several calls:

MCALL - fetch a segment and perform a subroutine jump
MEXIT - return from a subroutine called by MCALL
WSEG - write a segment back to the segment file
REENT - fetch a "reentrant" segment
SREEN - fetch a "reentrant" segment, save "shadow pages"

REENT and SREEN are described in the next chapter.

In many cases, segment switching is functionally equivalent to the use of overlays in background programs or reading data blocks from a file. The speed of segment switching is at least an order of magnitude faster; even if the segment must be read from the segment file, the file system is bypassed. The internal data structures contain all the information to make the lookup of the disk page as fast as possible.

If a segment is still in memory after previous use there is no need to read from the disk. A segment is not removed from memory until other segments claim its space. If physical memory is large and the activity on the system low (or mainly small segments used), one program may effectively have segments totalling several times the virtual addressing range concurrently in memory.

14.1 Calling a subroutine on a different segment

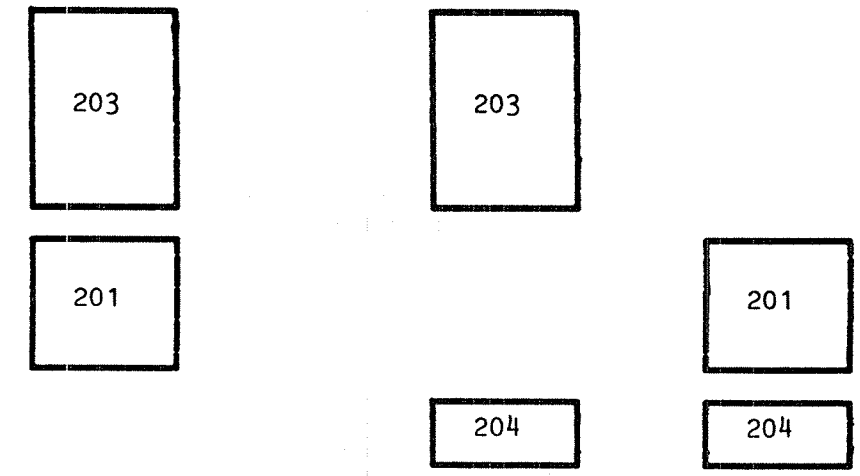
The MCALL (MON 132) call is used when a subroutine on a different segment is called. It replaces one of the current segments with a new one and performs the jump.

The call is only available in MAC and uses the T register (rather than the A register) to hold the address of the parameter list for the monitor call. The A register is free for use by the subroutine. (Standard Fortran call sequence uses the A register to point to the parameter list, in the same way as RT monitor calls.)

The monitor call parameters are the subroutine address and the new segment numbers. Usually only one segment is replaced, in which case 377B may be specified for the segment to be kept. (It is permitted to specify the segment number already present, if known.)

It is neccessary to know in which half of the word in the RT description the segment number to be kept is located. The first segment specified in a *NEW-SEGMENT command in the RT loader is in the right byte of the word, the second segment, if any, opened in another *NEW-SEGMENT (the default link segment during loading) in the left byte. If only one segment is opened, the left byte is zero.

Each time a segment is replaced, the segment (right or left byte) is explicitly specified, and it is the responsibility of the programmer to keep track of which segments are where in the word.




```
    203  201        203  201     ACTSEG in RT description
+   377  204    +   204  377     MCALL 2.nd argument
=   203  204    =   204  201     New segments in ACTSEG
```

One of the segment numbers may be zero. This is used for informing the operating system that the segment in the corresponding half in the RT description is no longer needed.

The two new segments may not overlap in the virtual address space; i.e. they may not use the same locations in the page tables. If the two segments use different page tables, no conflicts occurs.

MCALL causes the new segments to be fetched and the subroutine is started. The L register holds the return address and the T register the segment numbers of the calling program, the old value of the ACTSEG location in the RT description. This information is used on return from the routine. If the called routine uses the T or L registers, they must be stored by the subroutine if MEXIT is used for the return.

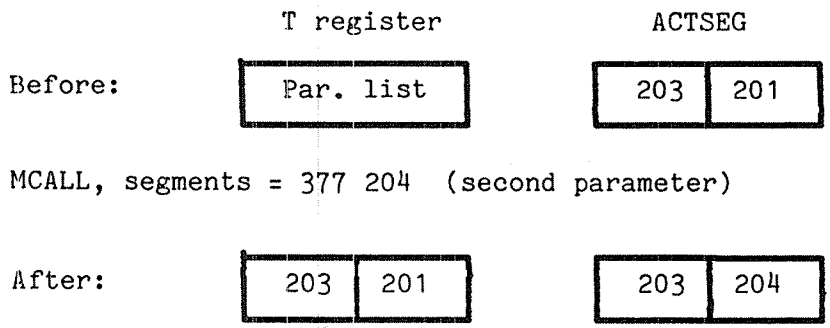


Fig. 25. ACTSEG and T register before and after MCALL

A call in MAC looks like:

```
        MCALL=132

        LDT (PARAM
        MON MCALL

PARAM,  SUBR
        (177603   % Replace segment in lower half in
                  % ACTSEG with segment 203
```

14.1.1 The register contents

All registers except T and L keep their values and may be used for parameter transfer to or from the called routine. If the A register points to the argument list (which is the case when Fortran is used), only one of the segments should be replaced. The parameter list and the parameters themselves must be in the non-replaced segment or they are inaccessible to the called routine (the corresponding addresses in the new segment would be interpreted as argument list and arguments).

14.1.2 The address of the called routine

The subroutine does not have to be located in the segment fetched, it can be in a non-replaced segment. If a subroutine has its working variables in a segment of its own, the ordinary data segment may be replaced with the private data segment while the routine is executing. When exit from the routine is executed by MON MEXIT (see below), the private data segment is removed and the old one brought back.

MCALL may be used for the sole purpose of exchanging the data segment, by calling a "virtual routine" at the location following the MON MCALL. This destroys the previous contents of the L register, but as long as the old contents are properly saved and restored, this is the fastest mechanism for virtually extending the data address space.

14.1.3 The page table

Subroutine jumps to another page table are not possible. Therefore, the called routine must be in a segment using the same page table as the current one.

If MCALL is used to replace a data segment with another one and the routine called is within the current segment (e.g. a "virtual routine" at the next program address), the segment may use another page table, addressed through the alternative page table mechanism described in section 3.7

14.1.4 The size of the segment

The two segments active at one time may not overlap if they use the same page table, but there are no limitations on the overlapping of the new and old segments.

The location where the MCALL instruction is executed may be replaced by another segment during the execution of the monitor call. There is nothing to prevent the new segment covering the entire 64K addressing area, replacing all the existing code;in that case all parameters must be transferred in registers.

14.1.5 MCALL nesting

As long as the return address, found in the L register when the called routine is entered and the old segment numbers, found in the T register, are saved in a the same way as on an ordinary routine call, MCALLs may be nested freely.

If the called routine is reentrant and the T register as well as the L register saved on a stack, even recursive MCALLs are permitted.

14.2 Returning from a routine on a different segment

Return from a routine called by MCALL is through the MEXIT call restoring the old segments and returning to the location following the MCALL.

Before calling MEXIT the T and L registers must have the same values as they had after the corresponding MCALL. After the MEXIT monitor call the T register contains the segment numbers of the present segments when MEXIT was called.

Example:

On segment 300B:

```
        MCALL=132
        MEXIT=133

        LDT   (PARLI        % T points to data element
        MON   MCALL
        .                   % Return here after MEXIT
PARLI,  SUBR                % Routine address on segment 200B
        (100300             % Get in segment no. 200B and 300B

        ...
```

On segment 200B:

```
SUBR,   STT   SAVT          % Entry point - save old segment
                            % numbers for the return
        COPY SL DA          % Save L register
        STA  SAVL
        ...
                            % Routine body

        LDA   SAVL          %
```

```
        COPY   SA DL        % Restore L and T register
        LDT    SAVT         %
        MON    MEXIT

SAVT,   0
SAVL,   0
```

MEXIT is available in the Fortran library as a subroutine. It allows only the segment in the left byte to be replaced, the second one opened during the loading process (the default link segment, see section 7.4.1). The Fortran MEXIT call is used for exchanging a segment only, no subroutine jump is performed. E.g. to replace the second segment with segment 212B, the call is

```
CALL MEXIT (212B)
```

The segment may contain both code and data.

14.3 Explicitly writing a segment to disk

If a data structure is partly updated when a (controlled or uncontrolled) system stop occurs, an inconsistent data structure may result. To take a checkpoint of a segment, modified pages may be written back to disk through the WSEG call.

In most cases, this call is only used as a safety feature. The RT monitor never overwrites a page in memory which has been modified without writing it back to disk first (except when connecting a reentrant segment).

More explicit control over when the writeback occurs may be obtained through the WSEG call. It is available as a Fortran subroutine:

```
CALL WSEG(302B)
```

or MON WSEG (MON 164) may be used in MAC. MON WSEG follows the RT parameter transfer standard:

```
        WSEG=164

        LDA (SEGN
        MON WSEG

SEGN,   (302
```

Even though the segment is written back to disk, it remains in memory until some other segment claims the space.

If external devices access the segment through DMA, the WIP bit in the page table is not set, and the page is not written to disk. This bit may be forced set for all pages in the segment as soon as it is placed in memory, by specifying the last parameter of the RT Loader command *NEW-SEGMENT, <WP/NP>, as "WP".

If "WP" was specified, the WIP bits in the page table entry remains set. If a DMA device makes further transfers to the segment these pages are written back before the pages are overwritten in physical memory by another segment.

If "WP" was not specified, WSEG will reset all WIP bits in the segment.

14.4 Loading a multisegment program

The RT loader can load to two segments concurrently, usually two segments which are active at the same time during execution. The first segment opened in a *NEW-SEGMENT command is termed the default load segment, the second the default link segment. The main program and possibly subroutines are located on the default load segment, subroutines and data shared with other segments on the default link segment.

Alternatively, the segments may be loaded one at a time. In that case, the subroutine/data segment is loaded first, specifying no link segment. There should be no undefined references on this segment.

14.4.1 The load commands

Either of *LOAD, *NREENTRANT-LOAD or *REENTRANT-LOAD commands may be used to load the main program and private subroutines loacated in the default load segment. The choice between them is determined as for a single segment system.

For loading code to the link segment, the second segment specified, the *LOAD command must be used, as this is the only load command that allows specification of the segment number. Remember that *LOAD changes the default load segment!

If the segments are loaded one at a time, either command may be used.

14.4.2 Linking between segments

A label to serve as a definition for references must be declared explicitly as a globally known label. Usually this is done by loading the definition of the symbol from a BRF file, but it may also be specified by the command *DEFINE-SYMBOL.

The two segments used at any time by a program may not overlap in virtual memory. The start address of one segment should therefore be above the highest address of all others that it is used together with. When two segments are loaded concurrently, the initial load address of the second one is 100000B, but if a different lower address is desired or the segments are loaded one at a time, the user must specify the load address through *SET-LOAD-ADDRESS.

14.4.3 Segment common

RTCOMMON is may be used as a common area for all segments on PT1. However, RTCOMMON is usually a scarce resource and any ring 1 program on PT1 may modify it.

A more easily controlled communication between segments can be obtained by using a separate segment rather than RTCOMMON. Segment space is usually available in abundance compared to RTCOMMON and only those programs linking to the segment have access to it.

The command

```
#SET-SEGMENT-COMMON <common label>
```

is analogous to #SET-RTCOMMON, but the common block is defined on the current link segment rather than in RTCOMMON. When the common block is loaded, a current link segment must exist - that is, two #NEW-SEGMENT commands must have been executed. (The #SET-SEGMENT-COMMON command may be executed before the second segment is allocated.)

The #PRESET-COMMON-ADDRESS may be used to force common blocks to be allocated on another address than the current load address. This is used if the segment contains both common blocks and ordinary code.

After definition, the common block is available to any segment that uses the same segment as a link segment.

14.4.4 Mutual references between programs on different segments

A program P1 on segment 270 may attempt to start program P2 on segment 271 - thus a definition of P2 must be available before segment 270 is completed.

If program P2 may start P1, P1 must be defined before segment 271 is completed. Thus, segment 270 and 271 must be loaded together, one as load segment, the other as link segment.

If either P1 or P2 requires that a third segment is loaded concurrently (maybe due to references to a third program), the system cannot be loaded directly in the RT loader, as it can handle a maximum of two segments at a time.

In such cases the command

```
#DECLARE-PROGRAM <RT name> (<RT description address>)
```

can be used to define a program before it is loaded. As references to a program are actually references to the RT description address, this allows one segment to be completed, having all its references satisfied. At a later time, the program may be loaded to another segment and the uninitialized fields in the RT description properly set.

The second parameter, RT description address, is rarely used. If specified, it must be the address of a free RT description. Default value is the first free description.

15 REENTRANT SYSTEMS

Several users may be using the same program or set of subroutines, concurrently. This is most notable with an editor or compiler; all terminal operators editing text or program essentially use the same code, all Fortran programs compiled at the same time use the same compiler.

As long as no modifications (store instructions) are made to the code, there is no reason why two users should not use the same set of instructions, the same physical pages in memory, rather than duplicating every instruction for every user.

If the same physical pages are used, more physical memory is available for other segments, there is less swapping and as each user of a common, reentrant segment make it the most recently used the probability of the segment being in memory (or the needed page within the segment) rises as the number of users increases. This reduces the number of page faults and consequently the number of process switches reducing CPU overhead as well.

15.1 The shadow page mechanism

Problems do not occur until one of the programs using the common segment modifies it - assigns a new value to a variable within the segment. Then the modification is effective for all users.

If the standard segment sharing is used, the common segment should contain read only data or routines and all variables should be located in the private segment. Many compilers, including the FTN compiler, have no other way of distinguishing between different types of data than using common blocks. This is sometimes cumbersome and does not easily interface to other languages. Also, special considerations are necessary be taken before compilation.

Alternatively, the segment containing the common routines and data may be declared as a REENTRANT segment with special properties, through the REENT call.

As long as no modifications are made, the entire segment is common to all users. When a user modifies a location, a private copy of that page is made. All later accesses go to the private copy.

Other non-modified pages are still common with other users. Private copies are only made when necessary to make users independent of each other.

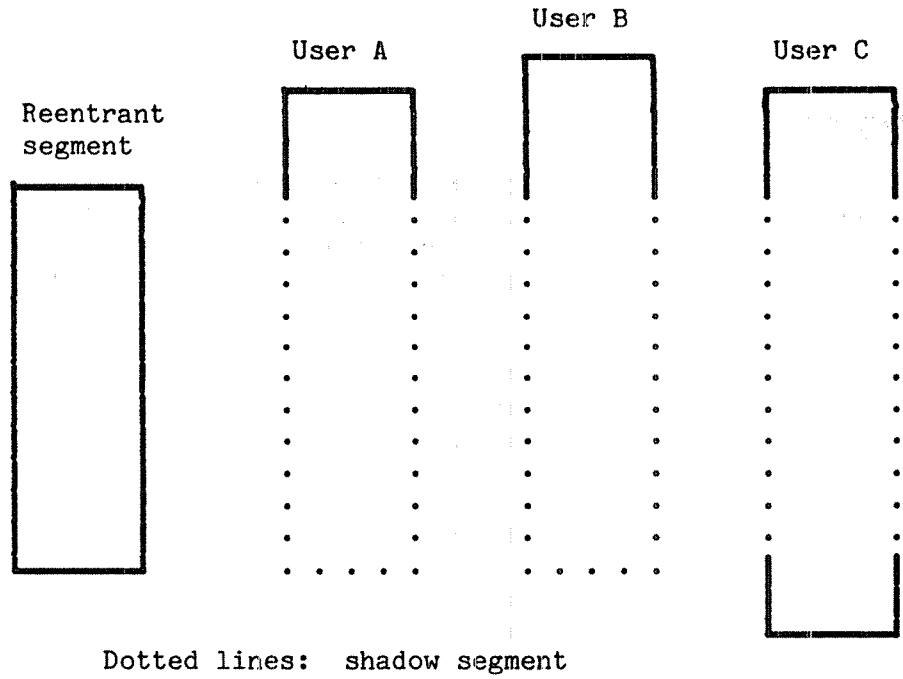
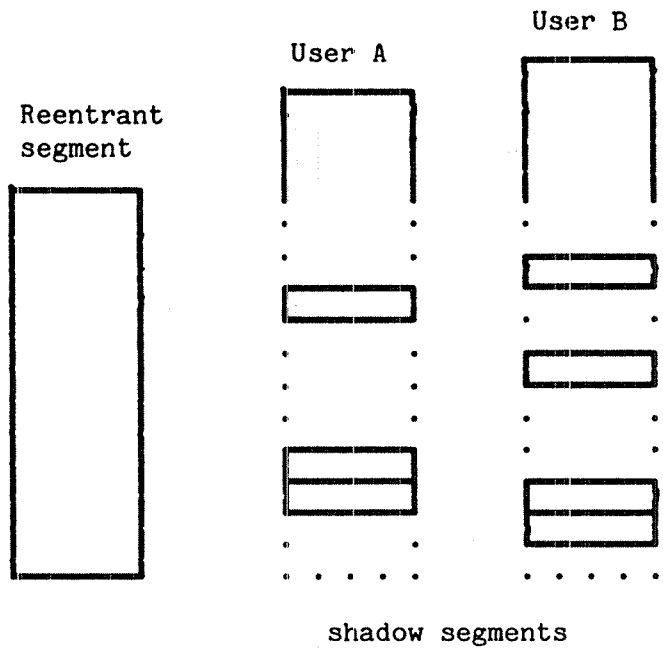


Fig. 26. Several users using the same reentrant segment

For administrative purposes the private copy must belong to a segment different from the reentrant segment. There must be a segment which is used for the private copies, covering the same addressing area as the reentrant segment.

The number of this segment, rather than of the reentrant one, is found in the ACTSEG location in the RT description. This segment is called a shadow segment, the pages in it are termed shadow pages. Each program using a reentrant segment normally has a different shadow segment. The number of the common, reentrant segment is found in the RSEGM location in the RT description.

The shadow pages are used if a store operation has been performed, the reentrant segment pages if not. The RT description contains a 64 bit wide bit map, one for each page in the logical address space. If the shadow page should be used the bit is set, otherwise it is reset.



Dotted lines: unmodified pages read from reentrant segment
Continuous lines: modified pages, private copy in shadow segment

Fig. 27. Modified pages are read from private copies

The reentrant segment may cover both segments, if two ordinary segments are used; then they both serve as shadow segments. A shadow segment may extend beyond the limit of the reentrant segment; in that area it functions as an ordinary segment.

If the shadow segment does not cover the entire reentrant segment, no modifications are legal in the area not covered. The reentrant segment may be read, but a store operation is trapped giving an OUTSIDE SEGMENT BOUNDS error, causing the program to terminate.

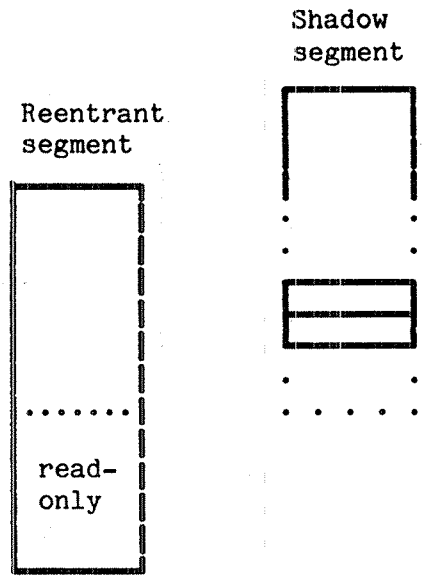


Fig. 28. Shadow segment is not needed for read-only pages

15.2 The REENT call

A segment is declared as reentrant through the MON REENT call (MON 167). REENT is available as a Fortran subroutine. Its only argument is the number of the reentrant segment and the standard RT monitor call parameter mechanism is used.

Fortran:

```
        CALL REENT(314B)
```

MAC:

```
        REENT=167

        LDA (PARLIST
        MON REENT

PARLIST,(314
```

Both declare segment 314B as a reentrant segment. If the segment number is illegal (out of range, not used or reserved, e.g. for loading), an error message is written on the error device and the program is aborted.

The REENT call is permitted in background programs (Sintran version E and later). SREEN (MON 212) is functionally equal to REENT, but will also write modified pages of the shadow segment to the segment file before the new segment is brought in (see section 15.9).

If another reentrant segment is already in use when the call is executed, it is no longer used but replaced with the new segment. The bit map is reset and all pages in the range covered by the new segment taken from the new reentrant segment.

For pages covered by the old, but not the new reentrant segment, data are taken from the shadow segment. Whether this is a private copy of the page in the old reentrant segment or the original page in the shadow segment (before the old reentrant system was declared), depends on whether any modifications were made while the first reentrant segment was active.

In cases where all pages in this area should be taken from the old reentrant segment, a dummy store operation (in Fortran this may be done by setting a variable to the value of itself) may be executed in each of the pages in this area, immediately before the new MON REENT is executed.

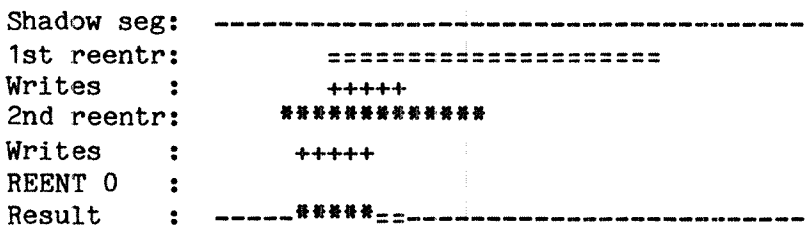


Fig. 29. Resulting effective pages after switching reentrant segments

The shadow segment must be a demand segment. This is not default when the segment is allocated and must be specified in the *NEW-SEGMENT command in the RT loader by giving the third parameter as DM.

Reentrant segments must be demand segments.

15.3 Optimizing the use of the reentrant mechanism

The reentrant segment mechanism may be used for all programs, whether the code is reentrant or not; the reentrancy is provided by the segment administration rather than by the program's allocation of space. The savings in physical memory requirements and swapping overhead are very dependent on the code generated.

Maximum gain is obtained if all working variables, written by the program, are gathered together in a small number of pages and all read-only data and code is located in other pages. If some pages are consistenly read-only and are located at the end of the shadow segment, the shadow segment need not cover these pages.

The code normally generated by the Fortran compiler in non-reentrant mode mixes code and data and the probability of having a page with code only is extremely low. Therefore, practically all pages are copied to the shadow segment after a short time.

If a Fortran program is compiled in reentrant mode ($REENTRANT-MODE compiler command), local variables are allocated on a stack and the code generated has no intermixed data. The variables that cause a page to be copied are spread over far fewer pages.

Pascal always generates pure code and uses stack allocation of data and is well suited to the reentrant mechanism.

Planc allocates local data on a stack, but data declared outside the outermost routine level are intermixed with the code. To make Planc suit the reentrant mechanism, all global data should be declared at the top of the module or imported from a separate data module.

MAC (including NORD-PL, which is translated into MAC) written in a traditional style is rarely suited for reentrant programs. ("Traditionally", local data is often located within the displacement field of the instruction.) All variable data must be declared together and addressed relative to the B and/or X registers. Literals allocated by the)FILL command are by definition read only, and may be used freely. Since the parameter list of RT monitor calls contain the addresses of the variables, it is usually read-only and may be located together with the program code, if desired.

15.4 Other use of the reentrant segment

Neither the reentrant segment nor the shadow segment should be used by any other program using these as ordinary segments.

If a "foreign" program (one not using the reentrant mechanism with this segment) makes modifications to the reentrant segment while it is being used by others, the effect on these other programs depend on whether they have their own copies (no effect) or not (modification applies). Debugging such systems is almost impossible.

A program using the shadow segment while the reentrant system is active may find an entire page suddenly replaced with one from the reentrant segment that was written to. This has a similar effect on the debugging as the above case.

15.5 The access bits of the segment

When a reentrant segment is fetched, the ring and page protect bits of this segment apply to the pages of the segment. A pure reentrant segment may be given read only or fetch only permit to prevent accidental write.

Pages in the shadow segment located outside the reentrant segment have the access protection of the shadow segment itself. As soon as a page is copied from the reentrant segment, it is considered a part of the shadow segment and its access is determined thereby. The operation causing the copying is restricted by the reentrant segment, while all future operations are restricted by the shadow segment.

Normally this causes no problems. However, if the shadow segment is read-only, while the reentrant segment allows write operations (and write operations are performed), the programmer should beware:

> WARNING! A write allowed page in a reentrant segment is, if a write operation is performed, copied to the shadow segment even if the shadow segment is read-only. This page is later written back to the segment file; the exact time is determined by the amount of swapping on the system. In this way, a read-only segment is modified!

15.6 Different page tables

The reentrant segment may use any page table, but if a page table different from that used by the executing code is used, there is no way to transfer control to the segment. It may be accessed as data using the alternative page table mechanism, described in section 3.7

The shadow segment must use the same page table as the reentrant segment. As when all segments use the same page table, no shadow segment is needed for the reentrant segment (or parts of it) that is not written to. If the reentrant data segment is read-only, e.g. error message text strings, no shadow segment is needed.

The data on a reentrant segment on a different page table must be accessed by using the ALTON call to set an alternative page table and setting the status register bit number 0.

15.7 Mixing MCALL/MEXIT and reentrant segments

If MCALL is used in systems also using reentrant segments, no segment switching is allowed that would affect the shadow segment mechanism. A segment used wholly or partly as a shadow segment may not be replaced through MCALL or MEXIT.

The segment fetched may not cover any part of the reentrant segment that has no shadow segment (necessarily a read-only part of the reentrant segment). It is legal to replace a segment not used as a shadow segment and the new segment may cover any area not covered by either the reentrant or the shadow segment.

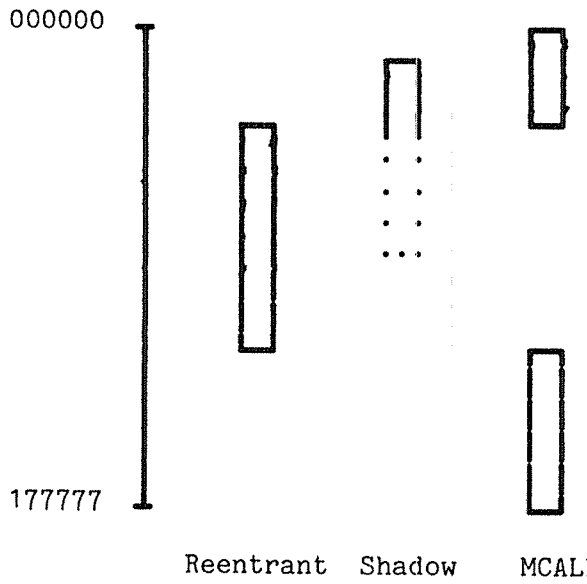


Fig. 30. The legal area for a MCALL segment

This also applies to the Fortran routine MEXIT. If the rules are violated, the calling program is terminated and an error message is printed on the error device (console).

A segment on another page table may be fetched by MCALL, without causing any collision. This may be used for data segments only, as there is no way to transfer control to another page table. The address of the called subroutine must be within the area covered by the reentrant or the shadow segment.

If a system no longer makes use of the reentrant segment but needs another it may disable the reentrant segment by calling REENT with segment number 0 as parameter. It may then replace the shadow segment with another one through MCALL/MEXIT.

15.8 The shadow segment after execution

All pages copied from the reentrant segment are a permanent part of the shadow segment and as the Written In Page (WIP) bit in the page table is set, the page is written back to the segment file.

The segment contains a mixture of pages, the original ones from the shadow segment before execution and pages copied from the reentrant one. For the next execution, the contents of the segment are unpredictable.

Pages outside the area covered by the reentrant segment still have contents explicitly controlled by the program and if the segment is a pure code segment which is never written into, the contents of the shadow segment are unmodified.

15.9 Automatic saving of the shadow segment pages

If a reentrant segment is declared, the pages in the shadow segment are not written back to the segment file, even if they have been written to. If these pages are later recovered by disabling the reentrant segment, the modifications may or may not still be valid, depending on whether the relevant page was swapped out on disk or not between the last modification and the connecting of the reentrant segment.

As an option, Sintran can be delivered with a monitor call that automatically writes all pages of the shadow segment back to the segment file before the reentrant segment is fetched.

Logically, this operation is almost equivalent to executing a WSEG call (see section 14.3) immediately before a REENT call. SREEN results in less administrative and disk overhead, as only the pages that have actually been written to and only pages in the area covered by the reentrant segment, are written back.

The call is available in MAC only and is used in the same way as the REENT call. To save the pages of the shadow segment and enter segment 203B as reentrant:

```
        SREEN=212

        LDA (PAR
        MON SREEN

PAR,    (203
```

SREEN is primarily useful when the reentrant segment (or a large contiguous part of it) contains pure code. When the reentrant segment is later disabled, the shadow segment is recovered with all modifications made before the reentrant segment was declared.

If modifications are made to the reentrant segment, the affected pages are copied to the shadow segment.

15.10 Disabling the reentrant segment

If MON REENT is called with a segment number of zero, no reentrant segment is used and any previously declared reentrant segment disabled.

The shadow segment is recovered. Whether the information in this segment is consistent or not depends on whether the reentrant segment was written to, as discussed in the two previous sections.

15.11 Multisegment reentrant systems used from background

The mechanism described above is used even for background processes when the program run is declared as a "reentrant subsystem" by user SYSTEM, using the Sintran command @DUMP-REENTRANT. In that case the background segment for the terminal is used as the shadow segment.

If more than one segment is needed, these may be written as separate programs and call each other using the COMND call (MON 70). This has a number of disadvantages; each set of subroutines, i.e. each segment, must be executable as a stand alone program. Also, the COMND call is slower than switching segments directly. This may be significant if there is frequent switching. Finally, the restart address (used when the @CONTINUE command is issued) is updated when COMND is used.

MON REENT may be used as a substitute. It is usually used in connection with two-bank systems, the reentrant segment covering the normal page table (or part of it). The alternative page table contains data for all the reentrant segments.

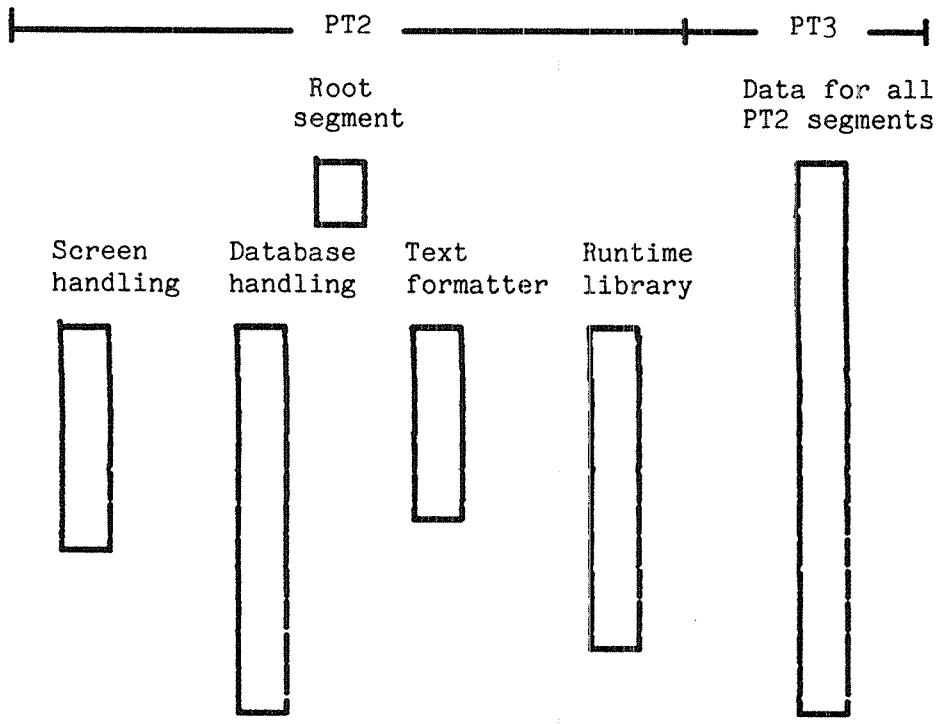


Fig. 31. Two-bank reentrant system for background

The "root segment" contains in the lowest part a routine to switch to another segment and jump to a routine in the new segment. Upon return from the routine, control goes via the root segment restoring the segment of the caller.

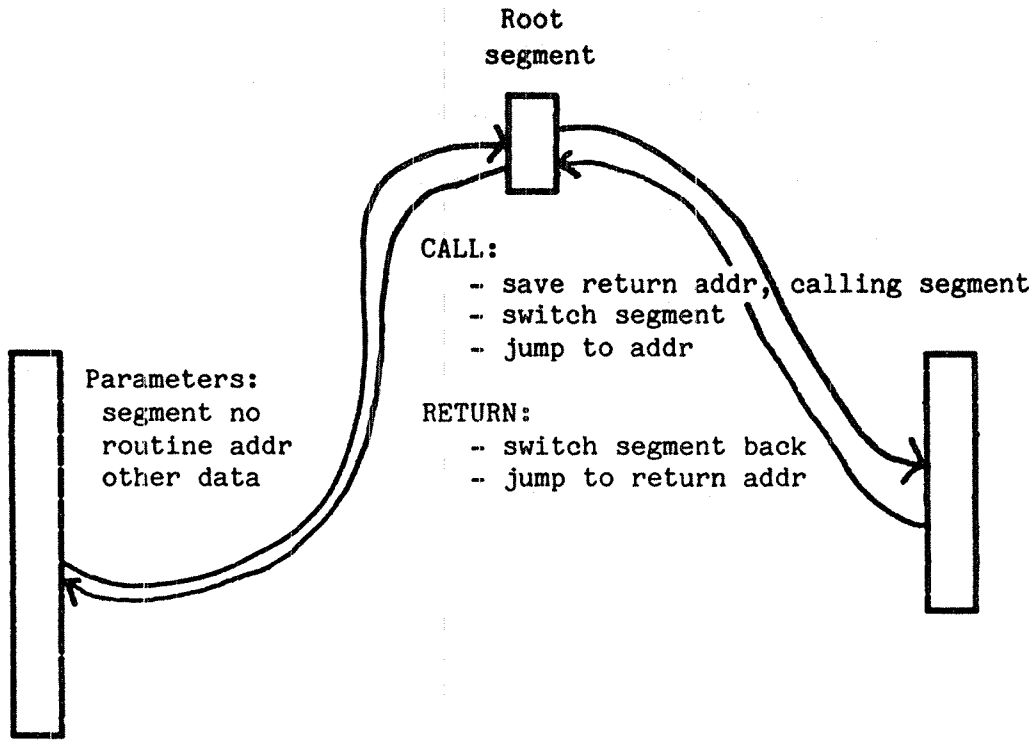


Fig. 32. Calling another segment via a root segment

The lowest pages of the root segment is usually present continously. Before any other reentrant segment is fetched, a dummy write should be made to all pages in this part to ensure they are copied to the shadow segment. Alternatively, these pages may be duplicated on all the reentrant segments.

MCALL/MEXIT calls are not available in background programs. REENT is not available in background programs in Sintran releases prior to Version E.

Reentrant segments for background programs usually use page table 2. Only if the alternative area (the data area) in two-bank systems is to be replaced should page table 3 be used. In almost all cases, the reentrant segment contains instructions rather than data.

15.12 Reentrant Fortran programs

Standard Fortran is not reentrant; a program or subroutine may not be reactivated before its previous execution has terminated. Attempts to reactivate a routine may be made by another program sharing the routine or by the same program calling a routine recursively.

This is because Fortran has static data allocation; all variables have fixed locations. This includes compiler generated variables, such as routine parameter locations and return addresses.

> If a program P1 calls a routine SUBR, the return address in P1 is stored in a system assigned location (called RETA). When the routine completes, a jump to the address found in RETA is performed.
>
> Before the routine completes, another program P2 calls SUBR. Now, the return address in P2 is stored in RETA. As soon as P2's call of SUBR is executed, a return jump to the address in RETA is performed (as expected).
>
> Now, P1 continues execution and completes SUBR. A jump to the address found in RETA is executed. This is no longer the address in P1. Rather, control is transferred to the return address in P2.

If the return address in P2 is valid in the segment (P1 and P2 could be on different segments, covering different extents of the address space), execution continues. This could be the worst thing to happen; maybe the program "crash", maybe it simply returns erroneous results that are not identified as such. It would in fact be better if the address was illegal in the segment of P1 and a fatal error occured!

The Fortran compilers (FTN, FORTRAN-100) may optionally generate reentrant code to prevent this problem arising. This is done if the command

```
$REENTRANT-MODE
```

is issued to the compiler before the program or routine is compiled.

15.12.1 The use of a stack

Unintentional modification of locations is not restricted to return addresses. Every local variable suffers from the same problem; the side effects are even more subtle and may easily go unnoticed.

For the calling programs to be independent of each other, they need individual copies of all variables in a routine. To keep two programs separate, the areas could still be in different but fixed locations.

If the routine calls itself recursively (see section 15.14), directly or indirectly (A calls B calls A), several areas are needed even within one program. There is no way of knowing the maximum number of concurrent activations, so fixed allocation cannot be used.

Some kind of dynamic allocation must be adopted instead. Every time a routine is called, a new area to be used for local variables is taken from a scratch area. On exit from the routine, the area is released. When stack allocation is used, space is always taken from one end of a contiguous area, the stack, when a routine is called. Routines called within the current one use the locations following etc.

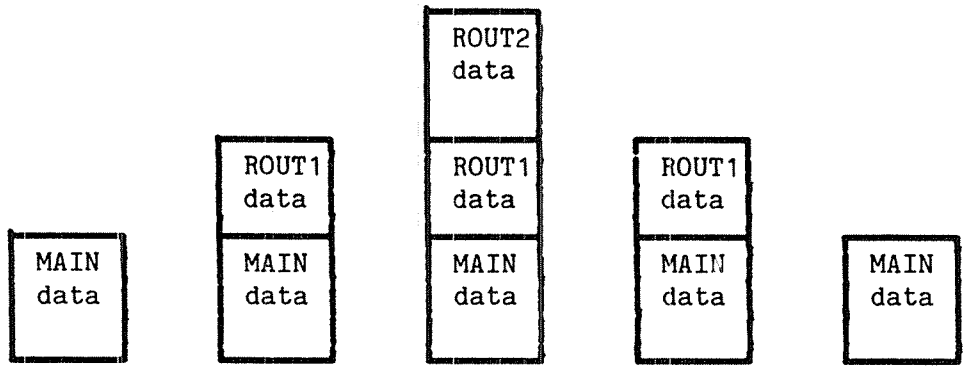


Fig. 33. Stack allocation

Main program	ROUT1 calls	ROUT2	ROUT1
calls ROUT1	ROUT2	completes	completes

Each program has its own stack. All data in one stack belongs to one program. The routine last called is always the first to complete. All reserved space is contiguous in one end of the stack area, all free space in the other end.

15.12.2 **Advantages of stack allocation**

From an RT point of view, the most obvious advantage is that each program is independent of other programs using the same routines. This is taken for granted by programmers used to background programming!

Many advanced algorithms make use of recursive techniques, section 15.14. Thus using iterative methods often results in more efficient programs. Using recursive alghorithms requires reentrancy, as provided by the stack mechanism.

Under some circumstances, the data space requirements may be significantly less when using stack allocation. Statically allocated data are continuously reserved, even if the routines using them are never active at the same time. When dynamic allocation is used, no space for data is initially reserved except the stack. The required size of the stack is no larger than needed by routines active at the same time. Routines called in sequence use the same data area.

15.12.3 Disadvantages and pitfalls of stack allocation

As the local data area used by a routine is reused by the next one called, the values assigned to local variables during one execution of a routine generally do not have these values initially when the routine is activated a second time. Some Fortran programs are written assuming that they do, and such programs need modification. Variables allocated in common blocks are statically allocated and will keep their values. (The SAVE statement is not implemented in Norsk Data Fortran. SAVE is implicit for all variables in non-reentrant Fortran and for variables in COMMON in reentrant Fortran.)

The user must estimate the amount of stack space needed. The FORTRAN-100 (ANSI-77) compiler reports the amount of stack space needed for one set of local data, but this must be multiplied by the maximum recursion depth of that routine. On the other hand, because not all routines are active concurrently, the space requirement is usually less than the sum of the space used for each routine.

Reentrant routines mode may not be used with a program compiled in nonreentrant mode. Using nonreentrant routines with a reentrant main program is possible. These routines must not be reactivated before termination by recursion or by another program.

Local variables may not be initialized. All local variables have undefined (random) values when the routine is entered.

15.12.4 Compiling reentrant Fortran

Reentrant code is generated by giving the command $REENTRANT-MODE before the $COMPILE command to the Fortran compiler. All programs and routines compiled after the $REENTRANT-MODE command until the compiler is left through $EXIT, allocate data in a stack fashion when executed.

Example:

```
@FORTRAN-100
ND-100 ANSI 77 FORTRAN COMPILER - 81.08.07
FTN:REENTRANT-MODE
FTN:COMPILE PRO4, "PRO4:LIST", "PRO4"
- CPU TIME USED: 1.1 SECONDS.  13 LINES COMPILED.
- NO MESSAGES
- CODE SIZE=134  DATA SIZE=0  COMMON SIZE=40  STACK SIZE=16
FTN:EXIT
```

The source program may not include DATA statements initializing local variables in subroutines and functions.

Recursive function and subroutine calls are now permitted. Be aware that this is an extension to the ANSI-77 standard. Recursive calls should be avoided if the program must be compatible with ANSI-77 standard Fortran. The (FORTRAN-100) compiler may optionally mark in the compiler listing Norsk Data extensions if the command $STANDARD-CHECK is given prior to compilation.

15.12.5 Reentrant Fortran and reentrant segments

Any routine or set of routines can be used by several users at the same time, provided the routines are placed on a segment used in connection with the reentrant segment mechanism.

Non-reentrant Fortran may be loaded to such a segment and used by several programs concurrently. But as code and data are intermixed, little or no space is saved in memory during execution, because practically all pages must be copied to the shadow segment. By compiling the program in REENTRANT-MODE, all code is located in one part, all code in another, making more efficient use of available memory.

The reentrant segment mechanism does not provide different data areas for recursive invocations of a routine. If recursive techniques are used, a Fortran program must be compiled in REENTRANT-MODE. This may but need not be used together with the reentrant mechanism.

15.13 Other languages and reentrancy

Most high level languages other than Fortran use a stack for local variables. The code generated is always reentrant and allows recursive techniques.

Like reentrant Fortran, they require explicit allocation of the stack. In Planc the stack is declared in the source program as a global (static) array, used by all routines including those shared with other programs. These other programs have their own global stack arrays.

In Pascal, the user defines symbols indicating the lower and upper limits of the stack at load time.

In MAC or NORD-PL explicit stack handling may be programmed or the stack handling may be duplicated from that generated by the Fortran compiler. This requires fluency in assembler programming and is rather cumbersome. As most assembler routines are tailor written to provide special functions for a Fortran (or Fortran compatible) program, a trick may be used to utilize the Fortran stack mechanism:

> Rather than allocating an area for local variables, the Fortran program provides an array that can be used for working storage by the assembler routine, as an argument. This array is allocated on the Fortran stack and thus reentrant. The assembler routine loads the base address of this array to the B register, and addresses all local variables B relative.

This technique may be used with any programming language, provided that an array address can be transferred as an argument. The actual parameter transfer mechanism may vary from one language to another. See appendix C.

15.14 Recursion

The solutions of some problems, in particular mathematical ones, are defined recursively - in terms of themselves. One characteristic example of this is the factorial function: N! (N factorial) is defined as N times the factorial of N-1 or, in mathematical notation, N! = N*(N-1)!

Another example is Pascal's Triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
  1   7   21  35   .   .   .
```

Each number in the triangle is the sum of the two numbers above it. The expression (X+Y)**N can be expanded to an N'th degree polynomial, where the coefficient of X**M * Y**(N-M) is the M'th number in the N'th row of the triangle - the top 1 considered the zero'th row. (Pascal's triangle has other interesting mathematical properties as well.)

Rather than calculating the entire triangle from the top down, any entry in the triangle can be calculated from the two above. These are calculated from the two above themselves, each calculated from the two above them again and so on until the top '1' is reached.

An integer Fortran function to calculate the M'th number in the N'th row would look like

```
      INTEGER FUNCTION IPASCAL(M,N)
      INTEGER M,N

      IF (N.LE.1 .OR. M.LE.1 .OR. M.GE.N+1) THEN
         IPASCAL=1
      ELSE
         IPASCAL= IPASCAL(M,N-1) + IPASCAL(M-1,N-1)
      ENDIF
      RETURN
      END
```

E.g. IPASCAL(3,4) returns the value 6, IPASCAL(6,10) returns the value 252.

The above solution is very simple compared to a purely iterative algorithm. IPASCAL calls itself to solve a subproblem (i.e. calculate the value of each of the two numbers above the current one). This is known as recursion.

Recursive techniques are very helpful if the data to be manipulated are hierarchically structured (tree structure), for network traversal, searching and sorting and a number of other important applications. Often the recursive solution is small, efficient and easier to understand than a corresponding iterative solution.

Indirect recursion occurs when a routine A calls a routine B and routine B calls A before it completes. Although split into two routines, the same situation appears - a routine is reinvoked before it has completed its previous execution.

If a recursive routine is to terminate, it must come to some point during execution where it completes without invoking itself. In IPASCAL this occurs if " N.LE.1 .OR. M.LE.1 .OR. M.GE.N+1 ": the value 1 is returned to the calling routine. Sooner or later this routine must complete and exit to the routine on the previous calling level and so on, until the main program receives its desired value. A proper termination condition is essential for the successful application of a recursive solution!

Recursion (direct or indirect) is not permitted in ANSI-77 Fortran, due to certain syntactical inconsistencies that result when subroutines are used as parameters. Recursion requires some kind of stack allocation, which in some machine architectures are difficult to implement without significantly increasing routine call overhead.

Norsk Data Fortran permits recursion in REENTRANT-MODE. This implies that the source program is not in accordance with ANSI-77.

The reentant segment mechanism does not provide sufficient reentrancy for recursive routines. For each recursive call - a call to a routine that has not yet completed - there must be a unique location to store the return address. A reentrant segment provides a unique location for each user, but each time a routine calls itself, the return address is overwritten by a new one unless a stack mechanism is provided. The program and all its routines must be compiled in REENTRANT-MODE.

- errors caused by hardware malfunctions

The error device is used for all system operator activities while the system is running. However, the microprogram communication can be run from terminal 1 only. If log in has been prohibited through the command @SET-UNAVAILABLE, log in is permitted from terminal 1 only, rather than from the error device.

Hardware errors may, as an alternative to being logged on the error device, be written to a segment reserved for this use (segment number 11B). User SYSTEM may use the SINTRAN command

```
@INITIALIZE-ERROR-LOG
```

to clear the buffer and cause future hardware error reports to be saved rather than printed. The command

```
@PRINT-ERROR-LOG
```

prints the error log on the specified file. (The buffer is not cleared by this command.) User SYSTEM can also obtain the device number of the error device through the command

```
@GET-ERROR-DEVICE
```

A program may use the call GERDV, MON 254. If the device is currently reserved by an RT program, the RT description address is also returned. If the error device is not reserved, D is zero. GERDV is available in SINTRAN version H and later.

```
GERDV=254

MON GERDV
STA RDEV      % A = device no
COPY SD DA    % D = reserving program
STA RTADR
```

An RT program can write a user defined error message on any terminal including the error device as long as the terminal has been reserved. Some routine libraries write to device 1, the console, regardless of the current error device. A message can be written to the error device without reserving it through the ERMON call or for reporting file system errors through ERMSG or QERMS calls.

16.2 Errors detected and handled by the RT Monitor

Appendix D in SINTRAN III Reference Manual ND-60.128 lists all the error messages issued by the RT monitor and gives brief hints about the probable cause of the error.

16.3.1 Error status conventions

With the exception of XMSG, all the systems mentioned above return a status value in the A register; XMSG uses the T register. In Fortran or other high level language, the status value is returned as an integer function value.

A status value of zero indicates successful operation. In assembler, the A register is cleared.

The return address after a monitor call is usually, but not always, the second location following the call ("skip return") if the operation was successfully completed and if the call is legal from background programs. If an error occurs, return is to the first location after the call.

For operations only allowed in RT programs, return is to the first location after the call regardless of whether an error occurred or not.

16.3.2 Writing a file system error message

The file system status returned to the program as an error code in the range 0:255 may be written to the error device using the ERMSG or QERMS calls. These are used exactly as in background.

The file system monitor calls all have "skip return" if no error occurs, return to the first location following the monitor call in case of error. ERMSG writes the error message and the program continues, QERMS writes the error message and terminates the program. The error text is given in SINTRAN III Reference Manual.

QERMS is not considered a serious error condition; a program terminating through QERMS executes as normal next time it is activated. This includes periodic activation and reactivation because the repeat bit (5REP) in the RT description has been set.

16.3.3 Writing a user defined error to the error device

A user defined runtime error code may be written on the error device through the ERMON call. The user may specify an error number in the range 50:69, plus a suberror number which may be any 16 bit signed integer. The format of the printout is

```
11.05.30 ERROR 55 IN KLOKK AT 16542; USER ERROR
SUBERROR: 10
```

The time of day (11.05.30), RT name (KLOKK) and program address (16542) as well as the error number (55) and suberror number (10) are reported. The suberror number is decimal.

MON ERMON expects the error number as two ASCII characters in the A register. The suberror should be an ordinary integer (binary) number in the T register.

```
ERMON=142

LDA (#55
SAT 12
MON ERMON
```

There is no error return from ERMON. The call is available in Fortran:

```
CALL ERMON(2H55, 10)
```

Both these calls generate the error message reproduced above. In FTN the "Hollerith" form must be used for the error number, either as a Hollerith constant as in the example above or as an integer written to, using an A2 format. A CHARACTER variable or string constant (enclosed in single quotes) is not accepted.

Example using an integer variable:

```
      PROGRAM MANE,40
      INTEGER IX(1)
      N = 57
      WRITE(IX,100) N
      CALL ERMON(IX,10)
100   FORMAT(I2)
      END
```

16.4 Errors resulting from SINTRAN or RT loader commands

The majority of RT functions available as monitor calls can also be executed as commands. While a program may receive an unclassified error code, a classified error code, provoke an error message on the error device or be terminated because of the error, the error reporting is in general poorer when commands are executed.

Some errors are common to RT and background programs, e.g. file system errors when an attempt is made to open a file through @OPEN-FILE (background) or @RTOPEN-FILE (RT).

A small number of error conditions particular to RT programs are reported. These include

- attempts to start a nonexistent RT program
- illegal/out of range parameters to @FIX, @PRIOR and commands relating to clock functions.

Not reported are errors in or unsuccessful completion of

- reservation of devices (@RESRV, @PRSRV)

- releasing of devices (@RELES, @PRLS)
- deletion or unintended modification of the segment file(s)
- erroneous parameters in @UNFIX

or in the RT loader

- setting the P register outside any segment in the *CHANGE-RT-DESCRIPTION command
- *EXIT-LOADER without *END-LOAD; this causes the RT loader to ignore all loaded code
- overlap between already loaded code and a file read by *READ-BINARY

Certain errors cannot be detected until the *END-LOAD command. In particular, the size of the segment (and consequently, the required contiguous area on the segment file) is not known until *END-LOAD.

The user is encouraged to verify that these operations are successfully completed. This can be done by inspecting the RT description, listing the waiting queue of a device (through @LIST-DEVICE) or executing related commands in the RT loader.

16.5 Monitoring error termination

Executing the SINTRAN commands

```
@DEFINE-TERMINATION-HANDLING RT <RT name>
@ENABLE-TERMINATION-HANDLING RT
```

will cause the specified program to be activated every time a (user) RT program error terminates: MON QERMS, MON ABORT or a fatal monitor error. This program may use RERRP to identify the terminating program, the cause of termination, and the program location when the error occured. These data may be used to prepare an error report, restart the program or take other appropriate action to bring the situation back to normal.

The RERRP call takes as its only parameter the address of a 6 element array. This array will return the following information:

- the RT monitor error code as two ASCII digits (without parity), see appendix D in SINTRAN III Reference Manual ND-60.128
- P register in terminated program when error occured
- error parameter 1, A register
- error parameter 2, T register
- RT address of terminated program
- RT address of program executing ABORT (if any) (zero if aborted by RT monitor due to fatal error)

Error parameter 1 and 2 are set by the error handling routine in SINTRAN, and may give further hints to the operation in progress. The contents are dependent on the kind of error.

For the last parameter to be valid, RT address of program executing the ABORT call, termination handling must be enabled. The first five parameters are set independent of termination handling, and may be read by any RT or background program. They are valid until the next error termination of an RT program.

On return from the monitor call, the A register is zero if the call was executed. The error code 153B in the A register indicates that the parameter address was illegal (outside segment etc.). There is no skip return.

```
        RERRP=207

        LDA (PAR
        MON RERRP
        JAF ERR      % Error in parameter address
         . . .       % Analyze error situation


PAR,    ERNUM

ERNUM,  0            % Will contain error code in ASCII
PREG,   0            % Will contain user program P reg
PAR1,   0            % Will contain error parameter 1
PAR2,   0            % Will contain error parameter 2
RTADR,  0            % Of terminated program
KILLER, 0            % RT address of ABORTing program,
                     %  0 if aborted by RT monitor
```

The error supervising program should have sufficiently high priority to guarantee that it will read and analyze one error before the next one occurs. This usually implies that it should have a higher priority than the programs it primarily handles.

Error termination handling may also be enabled or disabled through the call EDTRM, MON 206.

17 DIRECT TASKS

A direct task is a routine executing on one of the free interrupt levels 2, 5, 6, 7, 8 or 9 (level 5 only if XMSG is not included in the configuration), independent of the operating system. It is like running a job at a higher (software) priority, but the priority is controlled by hardware rather than by software. A direct task is started immediately when the interrupt level is activated, with no need to search execution or waiting queues.

Startup time is lower than for a high priority RT program on a fixed segment, because no page tables need to be loaded. This saves up five milliseconds.

Almost none of the facilities available to ordinary RT programs are available to direct tasks; these services are executed at a lower level and to activate the lower level the direct task must complete its operations - it cannot request a service and expect an automatic return when the service routine completes. Monitor calls, segment handling and demand paging are unavailable. If the task requires control of I/O equipment, it must explicitly issue all the required control instructions on the IOX level, writing the control and data registers of the various devices.

17.1 Activating a direct task

A direct task must be activated by an RT program setting the bit in the PID (Priority Interrupt Detect) register through the privileged MST (Masked SeT) instruction permitted only for ring 2 or 3 programs.

The task is active until it executes a WAIT instruction. Higher level activities may interrupt it, but lower level ones are not restarted before the direct task terminates through WAIT. This includes the RT monitor running at level 3. Even a direct task on level 2 halts the majority of RT monitor activities; parts of the monitor are executed at level 1 and they will not be executed.

Thus, if ordinary RT or timesharing activities are running concurrently with the direct task, the task should not be active for longer periods of time - in most implementations no more than a couple of seconds.

17.2 Implementing a direct task

17.2.1 Loading

The direct task is written as an ordinary RT program (but without monitor calls, possibly except XMSG) and loaded to a segment using the RT loader. The segment is then fixed in memory by the monitor call FIX or FIXC. This is because page faults must not occur in a direct task; these are handled at level 3, which cannot be activated while the task is running.

The load address must be set according to the page table locations used during execution (see below).

On page table 0 the available area starts on the first page following the memory resident section (the value of the symbol 7ENDC, found in PART-TWO (configuration dependent part) of the Sintran listing and ends at location 65777 (octal). The availabilty of page table 1 depends on the requirements of other RT programs in the system. Page table 2 is unavailable if background processes are active, page table 3 is available only if no 2-bank background systems are used or if a standard modification ("patch") has been done to make 2-bank systems use PT1 rather than PT3.

17.2.2 The ENTSG call

The direct task is declared by the ENTSG call. This puts the start address into the P register at the level on which the task runs. The PCR is set to the correct page table.

The contents of the page table used for the direct task are loaded at the time of the call. The PT locations are thereafter unavailable for other use (until the task has completed). The page table used should be one not used for other segments active at the same time. If background 2-bank systems are used, these employ page table 3 for the alternative (data) area.

Unintentional modifications to the page table made between the issuing of the ENTSG call and the actual activation of the task are not detected and may cause errors which are very hard to detect.

The page table of the segment need not be the same as the page table specified in ENTSG; the latter is used when the task executes, but other RT programs may access the segment before and after execution through the page table of the segment.

The protect setting may be different for these two tables; the ENTSG call sets the RPM, WPM and FPM bits, allowing the direct task all access to the segment. Protect violations, although detected through a level 14 interrupt, are handled at level 3, which is inactive while the task is running. Thus, enabling protection mechanisms would be meaningless.

Direct tasks may also be located in the paging off (POF) area and run with the memory management system turned off, making it independent of the contents of the page tables. The segment should be fixed (FIXC) at an address within the lower 64 K words of physical memory and page table 0 specified.

17.3 Communicating with the direct task

The task may be active and no ordinary RT program running or started until the task is terminated. Or the task is terminated and will not run until explicitly restarted.

This makes communication extremely simple. The RT program sets up a data area before the task is started and can read the result "immediately" after the activation. The entire task executes between the setting of the bit in the PID register and execution of the next instruction in the RT program.

As an alternative, the XMSG system may be used. Although the task may run at a level higher than XMSG (level 5), this monitor call is handled differently.

The task issues MON XMSG, which activates level 14. The call is identified and level 5 (XMSG level) enabled. Level 5 is not active until the task executes a WAIT instruction. XMSG always has a skip return if called from a direct task and activates the level from which it was called. (It is the explicit enabling of the calling level which makes it possible to call XMSG from higher levels.) A WAIT usually follows the XMSG call and return is to the first location after the WAIT:

```
 . . .              % Load function codes and parameters
MON XMSG            % Issue call - level 5 not yet active!
WAIT                % Activate level 5
STA STATUS          % Return from level 5 - skip return
                    % from the monitor call
```

If WAIT is not the first instruction following the call, execution of the direct task continues until the first WAIT. After execution of the call, return is to the second location after the XMSG call regardless of where the WAIT was executed.

17.4 Device driver routines

A driver routine on interrupt level 10, 11, 12 or 13 may be entered to the operating system similarly to a direct task. However, datafields and entries in the LDN table must be established at system generation time and it may be necessary to enter the device into the ident table through the @SINTRAN-SERVICE-PROGRAM.

17.5 Calling RT programs from direct tasks

A direct task may call a subroutine within SINTRAN III to start an RT program. The number of RT programs which can be started simultaneously is limited and specified at system generation time. The facility is not included unless specifically ordered.

The RT program is started by use of the Sintran routine RTDIR. This routine is known to the RT loader and may be declared as an external symbol. A call to this routine in MAC is as follows

```
        LDA    (ELEM
        JPL  I (RTDIR        % SINTRAN III SUBROUTINE

ELEM,   RTPRG; 0;0;0;0;      % RTPRG = RT description address
```

The A register points to an element of 5 locations. The first is a pointer to the RT description of the RT program, the rest is a work area for RTDIR.

Since RTDIR is located on PT0 and the parameter elements are on PT0, this method can only be used if the direct task is also on PT0. By using a system routine on level 14, it can be used from other page tables, e.g. PT3:

```
% Code on page table 3
    IOF
    LDA   RTDSC          % Pointer to RT description
    IRW   160 DT         % Set register on level 14
    LDA   (PPRTD         % Address of routine on level 14
    IRW   160 DP         %
    LDA   (40000         % Activate level 14
    MST   PID            %
    ION                  %
```

17.6 Activation of direct tasks from interrupts

A general purpose driver for activating direct tasks has been implemented. For each level there is a corresponding device number and a datafield. Device numbers:

```
440B - level 6
441B - level 7
442B - level 8
443B - level 9
```

Levels 5 and 2 cannot be activated by this method. E.g. if level 7 should be activated by a CAMAC interrupt, the monitor call ASSIG may set up the connection:

```
CALL ASSIG(441B,LAMX,ICRATE)
```

If a non-CAMAC interrupt should start a direct task then the datafield pointer must be entered in the IDENT and EXTEND table by use of the @SINTRAN-SERVICE-PROGRAM.

18 PERFORMANCE MEASURMENT AND STATISTICS

This chapter is a guide to analysis of response time problems of program systems running under SINTRAN III.

The analysis may been broken down into 4 stages:

* Clarification
* Measurement
* Diagnosis
* Solution

RT programs communicating with terminals are often used for transaction type systems, i.e. systems consisting of several terminals each running one or more application programs involving the use of one or more SIBAS systems. However, any system communicating with a terminal user may encounter similar problems, using databases or not.

18.1 Clarification

18.1.1 System characteristics

To find out why a given system does not perform as expected, a number of questions must be answered to clarify the problem:

- How much memory is available for swapping?
- How much memory is used by each application?
- How many of the terminals are running each application?
- What does a transaction comprise:
  - Which SIBAS calls and how many?
  - How many user instructions?
- How many SIBAS systems are running?
- What kind of SIBAS log is created, if any?
- How many pages for each SIBAS buffer?
- Are the transactions running under TPS?
- How many batch processors are running?
- What kind of tasks are being run under the batch processors and what are their priorities?
- Is the spooling system being used; if so, what is the estimated print volume per time period?
- Are any special RT programs being run; if so, what are their priorities?

18.1.2 Definition of response time

"Response time" is normally defined as

a) The time taken from the last user-typed input-character until the first character is output on the screen

or

b) The time taken from the last user-typed input-character until all characters of the response have been output on the screen.

The difference in time between b) and a) is often referred to as the "output time", which depends also upon the line speed of the terminal.

Norsk Data uses definition a), mainly because it enables a distinction between response time and output time. Also, it is more important to the user at his terminal.

18.2 Measurement

SINTRAN III has a number of measurment facilities which are described in detail in this section. "Standard" parameter values are suggested and hints given for the effective use of these tools. Their names are:

RT-PROGRAM-LOG
PROGRAM-LOG
HISTOGRAM
SYSTEM-HISTOGRAM
TIME-USED

In addition, under the Fortran compiler (FTN) there is a logging facility called

PROFILE-MAP

18.2.1 RT-PROGRAM-LOG

The RT-PROGRAM-LOG measures resource usage for a particular RT program and/or the system as a whole. It is also possible to measure the resource demands submitted from any background terminal, as each background terminal is internally connected to an RT program.

18.2.1.1 Preparation

Before running the @RT-PROGRAM-LOG, the background terminal from which the log is to be run (normally the system console, represented internally by the RT program BAK01) should be removed from the time slicer and its priority set higher than any other "ordinary" program. A priority of 60 (decimal) is enough if no special RT programs on a higher priority are present in the system. Priority is increased make the measurements as accurate as possible.

To do this:

1. Find the SINTRAN logical number of the terminal.
   (Hint: Use the TERMINAL-STATUS command. The terminal you are using may be identified from the command.
   If your SINTRAN version is E or later, the WHO-IS-ON command can also be used).

2. Execute the LIST-DEVICE command to find out which

background program has reserved that terminal.

3. Call up the SINTRAN-SERVICE-PROGRAM and remove the background program from the time-slicer.

4. Use the PRIOR command to set the priority of the background program.

Example:

```
@TERMINAL-STATUS,,

LOG.NO  USER          MODE     CPU-MIN OUT OF  LAST COMMAND
    1   SYSTEM        COMMAND        0      0  TERMINAL-STATUS,,
   45   INNKJQP-ES    USER           2    296  KOMP
  560   PRICING-PBO   COMMAND        2    387  WHO
  561   SIB2          RTWT           0    160  LO-FI
```

or

```
@WHO

===>      1  SYSTEM
         45  INNKJQP-ES
        560  PRICING-PBO
        561  SIB2
```

This shows that the terminal from which the TERMINAL-STATUS or WHO command was executed has the logical number 1. Now find which RT program has reserved it:

```
@LIST-DEVICE
LOG. UNIT: 1
INPUT/OUTPUT(0 OR 1): 0
RESERVED BY: BAK01
```

Then remove the terminal from the time-slicer (only user SYSTEM can execute the next two commands):

```
@SINTRAN-SERVICE-PROGRAM

*REMOVE-FROM-TIME-SLICE
LOG.UNIT NO.: 1
MEMORY? YES
IMAGE? NO
SAVE-AREA? NO

*EXIT
```

Finally, set the priority :

```
@PRIOR BAK01 60
```

18.2.1.2 Parameters for RT-PROGRAM-LOG

This example measures SIBAS (RT name: SIBA). If you just ont to measier overall system usage, give Carriage Return instead of the name of an RT program.

```
@RT-PROGRAM-LOG
RT NAME: SIBA
INTERVAL(SEC): 10
INTERRUPTS/SAMPLE: 1
LOG. UNIT NO.: 160
INPUT/OUTPUT (0 OR 1): 0
LOG. UNIT NO.: 161
INPUT/OUTPUT (0 OR 1): 1
OUTPUT FILE:
```

The INTERVAL is the time during which results of the sampling are accumulated before writing a report line on the log device. Use a short interval (5-10 seconds) if you are interested in rapid fluctuations. If you are interested in overall load, use a longer time (30-120 seconds). Default value (Carriage Return) is 60 seconds.

INTERRUPTS/SAMPLE: The log device is the "clock" for the RT-PROGRAM-LOG. Non-printing characters are output to the terminal (the reason why the cursor disappears on some screen terminals) and every "N'th" character a sample is taken of the current system state. "N" is the number of interrupts per sample, default is 8.

Below is an example of this:

The number of character interrupts per second from a 9600 baud terminal is given by 9600/11 = 873. The division by 11 is because it takes 11 bits to represent one character in the asynchronous protocol. Likewise, the number of interrupts per second from a 300 baud terminal (e.g. a Decwriter console) is given by 300/11 = 27.

Taking an INTERVAL of 10 seconds the number of samples taken during each 10 second period should be large enough to produce statistically significant results. Using 1 interrupt/sample for a 300 baud terminal produces samples at a rate of 27 samples/second. A 10 second period gives 270 samples, large enough to give reliable results. Using 2 interrupts/sample, the number of samples in the 10 second period would be half as many, i.e. 135; this is almost too small to produce reliable results. The calculation procedure is the same for all other line-speeds, only the numbers are different.

The overhead from a 300 baud Decwriter or about 27 characters/second terminal at 1 interrupt per sample is negligible (less than 1% of the CPU). A 9600 baud terminal sampling at maximum rate (873 samples/second) uses about 14% of the CPU.

An easy-to-remember way of selecting INTERRUPTS/SAMPLE is given in the table below. Divide the baud rate of the terminal by 300 and use this result as the number of interrupts per sample. This means that the sampling rate is about 27 samples per second:

For a 300 baud terminal, use 1 interrupts/sample
For a 1200 baud terminal, use 4 interrupts/sample
For a 4800 baud terminal, use 16 interrupts/sample
For a 9600 baud terminal, use 32 interrupts/sample

LOG. UNIT NO. may be used to identify one or two I/O devices whose usage is to be logged. If a device is reserved, it is defined as being in use. The devices often logged are the SIBAS internal devices. These are reserved by a user program when a SIBAS call is made and released when the call has been executed.

To measure SIBAS, the following logical device numbers (decimal) should be used:

	Input	Output
SIBA	160	161
SIBB	162	163
SIBC	164	165

18.2.1.3 Output from RT-PROGRAM-LOG

CPU	SWAP	FILES	DISC	PASSIVE	IO-WAIT	UNIT 160	UNIT 161
25/38	02/05	00/05	10	00	75	00/32	00/31

The first figure of each pair for CPU, SWAP, FILES and the two logical units shows the named RT program's percentage utilization of the corresponding resource. The figures for PASSIVE and IO-WAIT also refer to the named RT program. The second figure in each pair and the DISC figure show the total percentage utilisation of that resource.

In the above output, the CPU was used 25% by the given RT program and 38% by everything in the system, including the logged RT program.

The figures for SWAP mean that out of 100 samples, the given program was performing paging twice and some other program was paging 5 times.

FILES means "normal" use of files (user I/O). The DISC-figure is approximately the sum of the total SWAP- and FILES-figures, as these two are the only sources for the disk traffic.

All percentages are related to the INTERVAL-time given. If the INTERVAL is set to 10 seconds and the DISC-figure becomes 40%, then the disk system has been occupied 4 seconds out of 10. This includes time to set up transfers, seek-time and transfer-time.

The CPU, PASSIVE and IO-WAIT figures for a specific RT program add up to a certain percentage figure. (In the above example they are 25 + 0 + 75 = 100%). This percentage is equal or close to 100% if the load on the system is small, but starts dropping below 100% when the load increases. The missing percentages give an idea of the queue-lengths in the system. When an RT program is waiting for the CPU (or is in the

queue waiting to reserve some I/O-device), is not classified into any of the groups defined by RT-PROGRAM-LOG. So when a sample is taken when the RT program is in some device waiting queue, this will be a "missing" sample.

18.2.2 PROGRAM-LOG

The PROGRAM-LOG measures the relative amount of CPU used by each RT program in a given time interval. The log is started by giving the command START-PROGRAM-LOG which has one parameter, INTERRUPTS/SAMPLE. This parameter is the same as that used in the RT-PROGRAM-LOG command described in the previous section. To stop the log, STOP-PROGRAM-LOG must be given with the name of the output file for the results.

Example:

```
@START-PROGRAM-LOG
INTERRUPTS/SAMPLE: 30

@STOP-PROGRAM-LOG
OUTPUT FILE: TERMINAL

             PERCENT  SAMPLES
DUMMY          82       997
STSIN          00       0
RTERR          00       0
RTSLI          01       11
  .             .        .
BAK08          17       205
  .             .        .
```

In the time during which the log was operational, the program BAK08 was using the CPU in 17% and DUMMY in 82% of the samples.

DUMMY is the RT program which is the "current program" when no other program asks for execution (the system is idle). It has priority 0.

18.2.3 Histogram

The CPU HISTOGRAM measures the amount of CPU spent in different parts of the logical address space of an RT program. The parameters for the histogram are defined by the DEFINE-HISTOGRAM command. They are the name of the RT program, the start address of the logical area to be logged and the address interval. 64 equally sized intervals are logged. This means that the parameters choosen must satisfy the following equation:

```
START-ADDRESS + INTERVAL * 100B  <  200000B
```

The log may be started by the START-HISTOGRAM command and stopped by STOP-HISTOGRAM. The results are printed by PRINT-HISTOGRAM which has one parameter, the name of the output file.

Example:

```
@DEFINE-HISTOGRAM
RT NAME: GARP
START-ADDRESS: 0
INTERVAL: 2000B

@START-HISTOGRAM

@CC  Wait for a while...

@STOP-HISTOGRAM

@PRINT-HISTOGRAM
OUTPUT-FILE: TERMINAL

                    PERCENT    SAMPLES

OUTSIDE               0          0 OUT OF 12571
SYSTEM:               1          126
     0-  1777         0          0
  2000-  3777         7          880
  4000-  5777         9          1131
     .                .          .
176000-177777         0          0
```

"SYSTEM" here means CPU-time spent on hardware level 1, protection ring 1, 2 or 3, i.e. monitor call code is executed.

18.2.4 SYSTEM-HISTOGRAM

The SYSTEM-HISTOGRAM command measures the relative amount of CPU time used within sections of the physical memory on a specified interrupt level. Interrupt level, start address and size of the increments of the memory area to be logged can be defined by the command DEFINE-SYSTEM-HISTOGRAM. 64 equally sized intervals are logged. The log can be started by the START-HISTOGRAM command and stopped by STOP-HISTOGRAM. The results are printed by PRINT-HISTOGRAM which has one parameter, the name of the output file.

E.g. if the point of interest is CPU use on level 1 and in the address area (physical) 2000B to 3000B, the following command sequence should be used:

```
@DEFINE-SYSTEM-HISTOGRAM
LEVEL: 1
START-ADDRESS: 2000
INTERVAL: 10

@START-HISTOGRAM

@STOP-HISTOGRAM

@PRINT-HISTOGRAM
OUTPUT FILE: TERMINAL

                        PERCENT    SAMPLES
OUTSIDE:                  03         127 out of 3868
2000-2007                 00         0
2010-2017                 00         0
 .    .                    .          .
2740-2747                 97         3740
 .    .                    .          .
```

18.2.5 **TIME-USED**

The TIME-USED command (or TUSED call - MON 114, available in background only) returns the sum of the CPU-time used in INBT/OUTBT routines (hardware level 4) and in the user program itself (hardware level 1, protection ring 0) since the terminal logged on or batch job started.

It is important to notice that this is never the total CPU-time used, since time spent on levels 14, 13, 12, 11, 10, 3 and 1 (ring 1, 2 or 3) is not included. I.e. operating system overhead (apart from monitor calls for certain character handling) is not included in TIME-USED.

Here is a list of the activities on the 16 different hardware levels (for more details, see chapter 4):

Level	Activity
0	The system is in the idle-loop
1	RT programs and most monitor calls execute on this level
2	Not used
3	The kernel of SINTRAN III executes here
4	INBT/OUTBT monitor calls
5	XMSG/Not used
6	Not used
7	Not used
8	Not used
9	Not used
10	Driver routines for character device output
11	Driver routines for mass storage devices
12	Driver routines for character device input
13	Updating of real time clock
14	Internal interrupts (page faults, power fail, ..)
15	Not used

The fraction of CPU-time not included in TIME-USED varies from program to program and is often 1 - 30% of total CPU-time used. If certain monitor calls for character-handling are excluded, the more monitor calls a program executes, the bigger the fraction of CPU-time not included in TIME-USED.

18.2.6 **PROFILE-MAP**

The Fortran PROFILE-MAP is a table showing the number of times each statement in a Fortran program has been executed. The command PROFILE-MAP must be given to the compiler, with the name of the file receiving the output. A program using this facility must terminate at an END statement, not a STOP. The overhead in memory size is about 4 words per statement and execution time is considerably increased.

This command is particularly useful during program development and debugging, when parts of the program may be executed in background.

Example:

```
@FTN
NORD-10/100 FORTRAN COMPILER 2090H
$PROFILE-MAP
OUTPUT-FILE: TERMINAL
$COMPILE PROGRAM,LIST,OBJECT
53 STATEMENTS COMPILED
CPU-TIME USED IS 1.8 SEC.
$EXIT

@NRL
RELOCATING LOADER LDR 1935E
*LOAD OBJECT
FREE: 015233-177777
*RUN

  1 EXEC.   0 TIMES
  2 EXEC.   0 TIMES
  .  .      .   .
 48 EXEC   17 TIMES
 49 EXEC   17 TIMES
 50 EXEC    0 TIMES
 51 EXEC   17 TIMES
 52 EXEC    1 TIMES
 53 EXEC   18 TIMES
```

The FORTRAN-100 (ANSI-77) compiler does not have this facility, but a logging of lines or routine calls, without automatic counting of the total per line, can be obtained by using the Symbolic Debugger.

18.3 Diagnosis

The problem may be clear after one run of the RT-PROGRAM-LOG; e.g. the CPU load is approaching 100% or paging is permanently higher than 10 - 20%, the direct reason for bad response times may be obvious. Theoretical calculations should be made when possible, and compared with the measurements. Large discrepancies between actual and theoretical results should be questioned, since a fairly trivial misunderstanding may be the cause of the problem. E.g. a bad choice of break strategy may result in the CPU load being up to 60% higher than necessary.

A swap-rate higher than 10-20% (which means that too little physical memory is available) is often the most critical factor in an overloaded system. The swapping mechanism consumes both CPU and I/O resources and disk utilisation may exceed 30 - 35%, which is often considered a critical limit.

Very high utilisation of the CPU (70 - 100%) does not necessarily mean the situation is disastrous. On the contrary, if the different tasks in the system have priorities according to their importance (see next section) this can sometimes be an ideal situation.

18.4 Solution

Scarcity of some resource(s) will indicate that the computer system isprunning badly. There are usually only two ways to solve this kind of problem :

1. Add more of the resource to the system
2. Use less of the resource

The first option means buying more hardware. The second one involves finding out where the resource is being used up, i.e. in which programs and where in those programs. When the program area has been isolated, the choice is:

1. Optimising the program with respect to the resource
2. Lowering the relative priority of that program and accepting that it will run more slowly
3. A combination of 1 and 2.

18.4.1 Priority of batch processors

If the batch processors are being used to run CPU bound jobs, lowering their priorities is often a good solution to improve on-line response times. This can be done by the procedure described below. (Assume that the system has 2 batch processors.)

Abort the batch processors whose priority is to be lowered:

```
@ABORT-BATCH 1
@ABORT-BATCH 2
```

Be sure that the batch processors are idle when this is done.

Remove the batch processors from the time-slice:

```
@SIN-SERV-PROG
*REMOVE-FROM-TIME-SLICE 670D Y N N
*REMOVE-FROM-TIME-SLICE 672D Y N N
*EXIT
```

The batch-processors usually have logical numbers 670,672,674,.. This can be checked by the WHO-IS-ON command, which gives the batch-processors at the end of the list. If the SIN-SERV-PROG is to be used from a mode-file, use the "@"-character in front of all commands inside it.

Set the lower priority:

```
@PRIOR BCH01  8
@PRIOR BCH02  9
```

Any priority between 1 and 15 can be chosen and ensures that all interactive activity is on a higher priority than any batch activity. The RT names of the batch processors (BCH01, BCH02..) can be found by using the command LIST-DEVICE. (@LIST-DEVICE 670,, and @LIST-DEVICE 672,,) An additional means of tuning could be to append small batch jobs to the highest priority batch processor and big jobs to the lowest priority batch processor. Running two batch processors tend to cause time consuming access conflicts; if possible only one bath processor should be active at a time.

Restart the batch processors:

```
@BATCH
@BATCH
```

This procedure must be executed each time the system has been stopped and is best put onto a mode-file. The mode-file cannot be executed from the load-mode file, because it would try to abort the batch-processor under which it is running. Normally, it is started manually when the batch processors are idle.

When running this mode-file, timing reasons require a @HOLD must be inserted at three places in the file. Use the format @HOLD 5 2 (a 5 seconds pause) and place it:

1. As the first command in the file (SINTRAN ignores this first HOLD under certain conditions)
2. In front of the @SIN-SERV-PROG command
3. In front of the @BATCH commands

The effect on system performance of lowering batch processor priority depends largely on the kind of jobs executed in batch mode.

WARNING:

The procedure above should be executed **only** if the batch processors run CPU bound jobs, making little or no use of the file system and devices that need reservation.

If the jobs make use of the file system, performance is often degraded. The degradation occurs when low priority jobs reserve an internal file system semphore, but then looses the CPU to a higher priority job. This job may need the file system, so it enters the waiting queue. In the meantime another high priority job may have entered the execution queue.

Before the low priority batch job gets a chance to complete file system operations and release the semaphore, a high number of process switches may have taken place, particularly if there is a lot of terminal activity on the system; each time a break character is read from a terminal, it generally means that the background program is entered in the execution queue.

The only safe way to determine whether performance is improved in a particular implementation is to perform actual measurements under normal system load.

18.4.2 Priority of time-sharing terminals

Individual terminals can be permanently put on a lower or higher priority. E.g. a system has 15 terminals connected. 3 of them are used for program development and the rest for database transactions. The database activity is considered more time critical than the program development. The solution is to remove the 3 terminals from the time-slice and lower their priority (give them priorities between 1 and 15). This can be done permanently by executing the necessary commands (section 18.2.1.1) from the load-mode file.

18.4.3 Using reentrant systems

A high swap rate can be reduced by adding more memory; this is expensive, and the NORD-10 can under no circumstances handle more than 512 Kbyte. If most users use the same program systems, the reentrant subsystem mechanism may be used in background as well as RT.

The mechanism used in background is the same as described in chapter 15, but the segments are created from :BPUN files by the Sintran command

```
@DUMP-REENTRANT <name> <start addr> <restart addr> <file name>
```

This command is reserved for user SYSTEM.

Most standard subsystems (compilers, editors etc.) are delivered in :BPUN format ready for dumping, and the information sheet following the floppy informs about the start and restart addresses. Any installation may create :BPUN files of their own programs in the same way as :PROG files are made in the NRL loader. Rather than using the NRL command *DUMP, the command *BPUN is used.

*BPUN accepts numeric start address or the name of a defined symbol; the main entry point is equal to the program name. BOOTSTRAP ADDR is relevant for stand alone execution only. The default file type of the output file is :BPUN.

@DUMP-REENTRANT accepts numeric start and restart addresses only; the restart address is used by the @CONTINUE command.

To make maximum use of the reentrant mechanism, the program should be compiled in $REENTRANT-MODE. The reentrant FTNRTLIBR must be loaded and the symbol STEND defined equal to the uppermost FREE: limit.

```
@NRL
RELOCATING LOADER LDR-1935H
*LOAD MAINPROG, SUBROUTINES
FREE: 046664-177777
*ENTRIES-DEFINED
  MANE=022217      SUB1=022300      SUB2=034266
FREE: 046664-177777
*LOAD FTNRTLIBR
FREE: 062134-177777
*DEFINE STEND 177777
*BPUN
FILE NAME: "OWNPROG"
START ADDR: MANE
BOOTSTRAP ADDR:
*EXIT
@DUMP-REENTRANT
NAME: OWNPROG
START ADDR: 22217
RESTART ADDR: 22217
FILE NAME: OWNPROG
```

The reduction in swapping obtained by using reentrant subsystems rather than :PROG files depend on the number of concurrent users of the same system, the code/data ratio, whether REENTRANT-MODE was used and how often the system is used. Even if the reduction in swapping is moderate, startup is much faster and the administrative work much less for a reentrant system.

A high FILES percentage in RT-PROG-LOG may be improved by optimizing each program's use of files. This is discussed in section 13.13

19 DEADLOCKS

The term deadlock describes a situation where a program is waiting for some resource A and at the same time holding a resource B needed by the program reserving A. The second program cannot continue and the first program is never granted A.

A classical example is two programs P1 and P2 needing the card reader and printer. P1 starts execution and reserves the printer. P2 starts at the same time and reserves the card reader, then it tries to reserve the printer. But P1 does not release the printer until it has received the reader and completed. They are holding up each other eternally!

This example plus a lively imagination indicate why the term deadly embrace is sometimes used rather than "deadlock"...

A deadlock may involve several members: P1 waits for P2, which waits for P3, which waits for P4, ...which waits for P1.

19.1 Fatal deadlocks

Under some circumstances, the deadlock causes the RT monitor to go in an endless loop checking whether the device is available or not, stopping all other activity on the machine if nothing interrupts it; in general, no user RT program may interrupt the monitor.

The "brute force method" of resolving deadlocks is to push STOP, MCL (Master Clear) and LOAD on the computer front panel. This is not particularly desirable. There are more elegant methods so that terminal users do not lose their work and batch processors do not stop. These involve decoding octal dumps of RT descriptions etc. and access to the computer, in some cases as user SYSTEM.

This chapter describes some of these methods.

19.2 Non-fatal deadlocks

While fatal deadlocks may be related to weaknesses in the operating system, the most common deadlocks are caused by erroneous and conflicting use of devices (internal, external, semaphores) in user programs.

This may mean a number of user RT programs are waiting in various waiting queues, but other programs continue their execution. Other users do not notice the deadlock. SINTRAN III provides aids for detecting and resolving the deadlock, but it may be necessary to use commands reserved for user SYSTEM; user RT has no way of inspecting an arbitrary datafield.

19.3 "Virtual" deadlocks

Sometimes it appears that a deadlock has occurred - programs never complete a well defined sequence of actions - yet the programs are constantly active. The tracing of such situations resembles deadlock detection, but the solution may be different.

This example illustrates one case of such deadlocks. It is taken from the SINTRAN operating system itself, but similar situations may occur in user programs.

> Two programs share a memory buffer, used for working data read from a file. Program A reads its data from the file and starts working. At regular intervals a supervising program S checks which other programs request the buffer. If A has occupied it for a long time and someone is waiting for it, it is forcibly taken from A (among other things, to prevent deadlock!) and given to a program B.
>
> B starts working, reading its own data into the buffer. If it does not complete within some time limit, B looses the buffer and it is given to the next in queue.
>
> This may be A, which has to start from the beginning because its buffer was destroyed. It may not finish this time either before it looses its right to the buffer to B. And if B cannot complete, A and B throw the buffer back and forth between them forever...

The timeout function provided by S prevents a program from reserving the buffer and forgetting to release it. The period of S is selected so that all normal operations completes within time. But with a combination of a long queue of buffer users, a slow disk, slow memory, no cache and no "fast-cycle" CPU on top of an operation already using most of the time slot, the probability of a virtual deadlock is significant.

In this case, increasing the length of the time slot by a few percent allows at least one of the two programs to complete within a time slot. After that there are no competitors for the buffer and the other program can complete undisturbed.

19.4 Resolving a deadlock

Deadlock is detected because a job does not progress. Depending on the priorities of the programs involved, it may cause a small number of programs to "hang", or the entire system may be locked. The first case, where terminals are still active, allows SINTRAN commands to be used; but if the entire system is dead the only solution may be using the microprogram commands.

To make the system operative, the deadlock must either be resolved through "diplomatic" channels or the locked programs stopped or inhibited in some way.

The general technique is to identify the device(s) needed for one program to complete and force their release by the PRLS call or by manually modifying the reservation link. Then the device can be reserved by the waiting program, which can complete and release all its resources. This should allow the other program(s) to continue.

19.5 Freezing active programs

In a busy deadlock, the programs execute but do not complete, it may be necessary to inhibit execution without terminating the programs. In case of a virtual deadlock, this may be sufficient to solve the problem, but usually it is only a method to freeze the situation in order to analyze the queues.

There are two essential methods for stopping a program:

- Setting the priority to 0
- Setting the program in an IOWAIT state

The priority can be set by user RT as long as the computer is available from a terminal; the @PRIOR command is used.

Setting the program in IOWAIT can only be done by user SYSTEM or through microprogram commands. If all terminals are inactive because the deadlocked programs have a higher priority than the terminals, the microprogram may be used to modify the priority.

To patch the RT description it is necessary to know its address. The address of the currently active program is found in resident location 10B.

Both the IOWAIT flag and the priority is found in the second word of the RT description (RT description address + 1), bit 17B and bits 7:0 respectively. Bits 10B:16B should not be modified if the program is to be restarted at a later time.

Given the RT description address of program A as 40321, the priority is changed from its current value of 80 (120B) to zero by the commands

```
@LOOK-AT RESIDENT
READY:
40321/      0  (cr)
  120 0  (cr)
    0 .
END.
@
```

It is a good idea to keep an up to date list of the RT programs in the system, obtained through @LIST-RT-PROGRAMS for use when the system stops.

The microprogram commands are similar, but do not give the prompt. No explicit termination of memory inspection mode (like the full stop in @SINTRAN-SEVICE-PROGRAM) is used.

19.6 Tools to search the queues

All queues in the system may have to be searched to resolve a deadlock. The queue descriptions in chapter 6 must be thoroughly studied before any modifications are attempted.

The queues are linked through link locations:

Time queue:	Resident 12B	(head)
	RT address + 0B	
Execution queue:	Resident 13B	(head)
	RT address + 20B	
Waiting queue:	Datafield + 3B	(head)
	RT address + 20B	
Reservation queue:	RT address + 23B	(head)
	Datafield + 0B	

If the 5IOWT bit in the RT description is set, the program is not executing even if the program is first in the execution queue. The currently executing program is pointed to by resident location 10B rather than 13B.

19.6.1 SINTRAN commands for user RT

These commands are only useful when SINTRAN is running and terminals can be used. This is often the case when ordinary programs are deadlocked waiting for devices.

The commands are described in chapters 6 and 10:

@LIST-RT-DESCRIPTION <RT name>

@LIST-DEVICE <log unit> <input/output>

Unfortunately, there is no predefined mapping from the datafield address to the logical unit. The listing of the configuration dependent part of SINTRAN (PART-TWO) is a great help. The datafield address can be looked up (the datafield address directly gives the left hand column location counter) and the name and type of field are written as comments.

E.g. if the program A does not progress, inspecting the RT description shows that it is waiting for some device whose datafield address is 33107. In a small program, the number of devices reserved may be low, giving one or two likely candidates. The program counter (P= 166) and the list of reserved datafields may provide clues to how far execution has progressed, localizing the device.

```
@LIST-RT-DESCRIPTION A
 RING:0 PRIORITY:     32
LAST STARTED:    8 MINS   42 SECS
START ADDRESS:      0, SEGMENTS:      0    270
P=    166
X=     72
T=      3
A=      0
D=    165
L=     24
S=      0
B=    255
WAITING FOR:  33107
ACTUAL SEGM.:      0    270
RESERVED DATAFIELDS:
  33103
```

SINTRAN PART-TWO (in this configuration) at location 33107 contains:

```
   033107  SEMI2, 0;0;*-2;0
```

These are the initial values for the datafield of the second semaphore (all semaphores present in the system are listed in the same section, under the header SEMAPHORE DATA FIELDS). As semaphores are numbered from 300, this one has logical device number 301:

```
@LIST-DEVICE 301B 0
RESERVED BY: B
WAITING RT-PROGRAMS:
A
```

An inspection of the RT description of program B reveals that it is waiting for the device reserved by A, datafield address 33103:

```
@LIST-RT-DESCRIPTION B
 RING:0 PRIORITY:     32
LAST STARTED:    8 MINS   59 SECS
START ADDRESS:     75, SEGMENTS:      0    270
P=    166
X=    160
T=      3
A=      0
D=    165
L=    114
S=      0
B=    343
WAITING FOR:  33103
ACTUAL SEGM.:      0    270
RESERVED DATAFIELDS:
  33107
```

The loop is closed, with only these two programs involved. The conclusion can be checked by looking up address 33103 in PART-TWO, or by deducing from the source program that this is the first semaphore in the system, device number 300 and verifying that B is in the

waiting queue:

```
@LIST-DEVICE 300B 0
RESERVED BY: A
WAITING RT-PROGRAMS:
B
```

When the deadlock is isolated, the loop must be broken by aborting either A or B or releasing either semaphore from the reserving program. In most cases the latter causes the program to error terminate when it attempts to access the (no longer reserved) device - in other cases the program simply "forgot" to release the device.

Even if the deadlock is resolved, this does not guarantee that the programs will progress normally! In the example above, the two semaphores may be protecting non-reentrant data areas and aborting either program (or lifting the protection) could leave the data structure inconsistent. This cannot be decided without detailled knowledge of the programs involved.

19.6.2 The SINTRAN-SERVICE-PROGRAM commands

The SINTRAN-SERVICE-PROGRAM is not available for user RT.

It does not provide more information about an RT program than the @LIST-RT-DESCRIPTION command, but gives it in a different format. This format may be more suitable as a starting point for inspecting resident memory; the addresses of the reservation queue and waiting queue link location is given directly as addresses in resident SINTRAN. This is helpful if e.g. a program must be unlinked from a waiting queue:

```
*DUMP-RT-DESCRIPTION
RT NAME: B MEMORY,,
OUTPUT FILE: TERMINAL

TLINK:          0
STATUS:      1100
DTIM1:          0
DTIM2:     161722
DTIN1:          0
DTIN2:          0
STADR:      24003
SEGM:       37003
WLINK:          0
ACTSEG:         0
ACTPRI:    100000
BRESLINK:   44250
RSEGM:          0
WINDOW:       126
RTDLGADD:  146450
DPREG:      17166
DXREG:          3
DTREG:          0
DAREG:        165
DDREG:          1
DLREG:        114
DSREG:       2000
DBREG:      12343
BITMAP:         0
BITM1:          0
BITM2:          0
BITM3:          0
BITM4:          0
BITM5:          0
BITM6:          0
BITM7:          0
*
```

Datafields can be inspected and modified by symbolic names, including inspection of device dependent subfield. The *CHANGE-DATAFIELD command is used:

```
*CHANGE-DATAFIELD 300B, I
MEMORY? Y
IMAGE? N
SAVE-AREA? N

BWLINK/40321 .
```

The RT description address of the first program in the queue waiting for the semaphore (logical device 300B) is 40321. To unlink the program B (above) from the queue, BWLINK must be changed to 33107.

If a slash (/) is typed immediately after the keyed in value, the value is interpreted as the address of a location and the contents of that location are immediately displayed. To advance to the next location a carraige return is typed. If a value is keyed in and followed by a carriage return with no slash, the current location is set to the specified value.

19.6.4 Microprogram communication (MOPC)

If a deadlock or an error has caused the machine to stop or hang, unaccessible through terminals, the microprogram communication (MOPC) can be used to inspect and modify main memory locations before the machine is restarted.

MOPC can be operated from the console only. It is available even when the CPU is running normally and can then be entered by user SYSTEM by the command @OPCOM. In most cases, however, other methods are more appropriate for resolving deadlocks.

In order to operate the MOPC in STOP mode (CPU not running) the front panel key must be turned to the ON position. The MCL (Master Clear) button should not be pressed; it will clear all registers and a complete restart (LOAD button) must be done.

The commands and their use closely resembles the @LOOK-AT command. A specified address is interpreted as a physical memory address and may be up to 24 bits (8 octal digits), provided corresponding physical memory exists. The lowermost 64K of memory is the SINTRAN resident and paging off area, where all RT queues and tables are found.

After the necessary patches have been made the CPU may be restarted by typing an exclamation mark or, on NORD-10, pressing the CONTINUE button.

No protection mechanisms are applied when patching memory locations though OPCOM.

19.7 Preventing deadlocks

Though some deadlocks can be resolved, it is preferable that they never occur. There are well known techniques for "diciplining" the use of resources; some are enforced by the operating system, but most are dependent on each user program adhering to certain rules.

Most techniques guarantee that deadlocks do not occur, but lead to poorer utilization of system resources; although a resource is available, it may be illegal to allocate it to a requester. It remains idle to guarantee its availability if an already active program needs it to complete.

Whether deadlock prevention techniques are used depends on the minimum permissible utilization of resources and the cost of a system stop (which may be the result of a deadlock).

19.7.1 Multiple reservation

If a program reserves all the resources it needs as soon as it starts execution, it is never caught midway through execution unable to complete. If one of the requested resources is not available, the program should not start, but try again later.

The same technique can be used for a group of resources used during part of the execution time only, but if other resources are used as well, deadlock is prevented only with respect to the resources in the protected group.

SINTRAN III does not provide a true multireservation facility (of the kind "Reserve all or none of A, B and C"); it must be simulated by the user program.

19.7.2 Using a semaphore

If the users of a set of resources are limited to a small and well defined group of programs, a semaphore may be used to protect the entire set; before a reservation of one of the elements in the set can be reserved, the semaphore must be reserved. And before the semaphore is released, all resources in the set must be released.

It is important that while the semaphore is reserved, no resources outside the set must be reserved. This may cause the program to enter a waiting queue and the owner of the device may be waiting for release of the semaphore. Although it is possible to specify a non-zero RETURN parameter to the RESRV call, preventing the program from entering a waiting queue, this should be condsidered poor programming practice.

19.7.3 Complete initial reservation

The use of a semaphore requires all programs to use the same technique and an erroneous reservation in one of them may cause a deadlock even if all the others are correctly programmed.

A program may reserve all the resources it needs in one batch independently of other programs, and give up the operation in case of failure. The complete resource requirement must be when the program starts; this is not always the case, e.g. user input may determine which files are to be used. When the requirements are known, the program may protect itself from entering a deadlock situation.

The resources must be reserved in sequence, one by one, but if the reservation of one fails, all resources already reserved must be released and another attempt made later. The RESRV call must specify a non-zero RETURN flag, preventing the program from entering a waiting queue.

The main disadvantage of complete initial reservation is that it ties up resources for an unecessarily long time if the resources are not used until the end of the program.

```
      PROGRAM RESALL, 60

      PARAMETER (INPUT=0, OUTPUT=1, IMMRETURN=1, SEC=2)

      IF (RESRV(RESOU1,INPUT,IMMRETURN).NE.0) GOTO 9999
      IF (RESRV(RESOU2,INPUT,IMMRETURN).NE.0) GOTO 9998
      IF (RESRV(RESOU1,OUTPUT,IMMRETURN).NE.0) GOTO 9997
      IF (RESRV(RESOU2,OUTPUT,IMMRETURN).NE.0) GOTO 9996

C     These are all resources needed by the program - now go on to

C           P R O G R A M    A C T I O N S


 9996 CALL RELES(RESOU2,OUTPUT)
 9997 CALL RELES(RESOU1,OUTPUT)
 9998 CALL RELES(RESOU2,INPUT)
 9999 CALL RELES(RESOU1,INPUT)

C     Retry after 10 seconds:

      SET(RESALL,SEC,10)

      END
```

19.7.4 Hierarchical reservation

If all programs agree on a specific sequence of reserving/releasing, better utilization of available resources is possible. All resources have a priority so that those reserved for short periods of time (e.g. a non-reentrant routine) is given high priority; those reserved for a longer have lower priority.

There are two main rules:

1) A program may only reserve resources of a higher priority than the highest one currently reserved

2) Before resources of a certain priority are released, all resources of higher priorities must be released.

The first rule does not prevent a program from reserving several resources of the same priority, but all must be reserved concurrently. If they are not all available, the program may enter a waiting queue for all devices required on that level, but not for a single one while reserving the others.

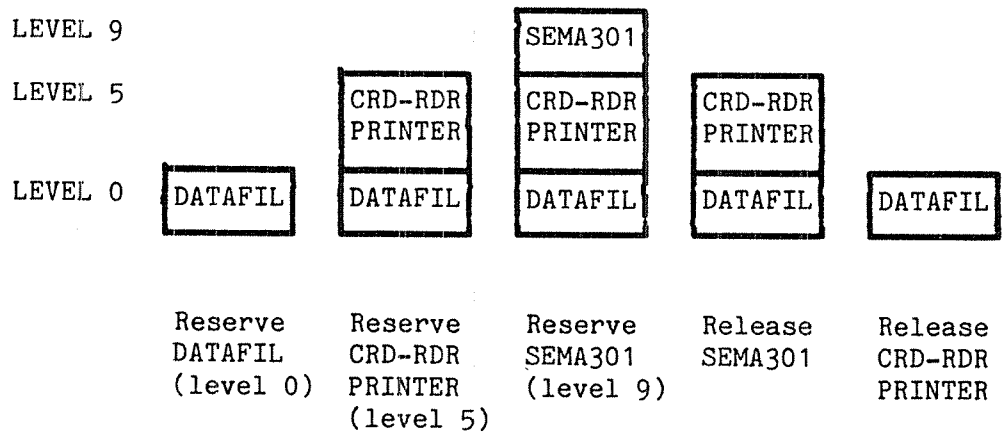


Fig. 34. Resource priorities

No resources other than those assigned a priority level should be reserved by a program while any such resource is reserved. This can easily be prevented by treating all devices as having some priority.

A priority is not assigned by SINTRAN, with one exception; the parameter list for @SCHEDULE lists devices in increasing number sequence. The device number is used as the priority level and there is only one resource per level.

In some RT systems the device number may be used, but generally an independent priority must be assigned. Each level may consist of one or more resources, but if a program needs several of the resources at one level, these must be reserved together, protected by a multi-reserve mechanism as described in the previous section.

Because all reservation on a given level must be made in one batch, a semaphore need only be reserved while the actual reservation is made. When a program has aquired a selection of resources on a given level, it is not permitted to request more resources on the same level, so continuous semaphore protection is unnecessary.

In fact one semaphore can be used for all levels. This means programs have to wait for other programs to complete the reservation process, but this is usually quite quick. The semaphore is reserved before reservation is started and released as soon as the resource(s) is acquired or found inacessible.

Hierarchical reservation depends on all programs adhering to the two rules above and the device priority must be unambiguously defined.

19.7.5 The banker's algorithm

Although a resource is needed some time in the future by a program PROG1, it may be allocated to another program PROG2 without risking a deadlock if there is a guarantee that PROG2 will complete. Such a guarantee may be given if PROG2 does not reserve any other resources or only request resources covered by a similar guarantee.

To determine whether PROG2 will complete, all PROG2's current and future demands must be known. This does not require that the resources are actually reserved! They are still available to other programs as long as these programs are guaranteed to complete.

The method used to determine whether PROG2 should be granted the requested resource is called "the banker's algorithm". When a resource is requested, a simulation of the situation after a (possible) allocation is made. If the simulation shows that the requesting program can complete, the resource is granted, otherwise it is temporarily denied, waiting for other programs to release the resources they are holding. It is safe to assume that these other programs eventually release their resources - otherwise the resources would not have been granted.

A coded solution and details of the algorithm can be found in several well known books on computer science. Below is a rough outline of the procedure.

boolean function: allocation_legal

```
assume resource is granted to requester
repeat until no more programs may complete
   for all active programs do
      if the program may complete,
        given the current (simulated) situation
      then assume all its resources released
      endif
   endfor
endrepeat
if all programs are able to complete
then allocation_legal := true
else allocation_legal := false
endif
```

Whether a program is able to complete or not is determined by comparing the available resources with the total resource requirement of the program, stated when the program is submitted for execution.

The banker's algorithm has two major disadvantages. The routine above must be executed for each reservation by any program and has a complexity of $mn(n+1)$, given m resource types and n active programs. I.e. if the number of active programs doubles, the worst case execution time for the routine quadruple. (If a program can request any one out of several equal resources, these are of the same type. An example in SINTRAN III is the spooling files.) The algorithm represents a very pessimistic view, assuming that any program may request all the resources needs at the same time. This is often not the case.

However, it does provide full deadlock prevention, and if the application requires 24 hours a day operation it may be worth its price.

20 ND-NET AND XMSG COMMUNICATION

The methods discussed in chapter 12 for communication between programs allows fast and well defined communication. They have several drawbacks, however:

- they do not provide communication between RT programs in different machines
- the maximum number of concurrent connections is limited, and fixed at system generation
- communicating programs must have agreed upon a specific communication channel to use
- communication with drivers and direct tasks is impossible
- the buffer capacity is small (byte oriented internal devices)
- the number of calls to transmit a longer message is (usually) high
- communication is strictly one-to-one

ND-Net and XMSG both provide multi-machine communication. For XMSG the number of simultaneous communication channels is high and not fixed at system generation time. A receiver can be identified by an alphanumeric name, and the communication system will take care of the actual routing of the message. From the communication point of view, a background program, RT program, direct task or driver look alike regardless of program type and in which machine they are located. XMSG is a many-to-one and a one-to-many communication system - a port may be used for communication with any number of other ports.

These communication systems are discussed in detail in the SINTRAN III Communication Guide (ND-60.134). This chapter will give a short presentation of how XMSG and ND-Net are used from RT programs.

20.1 ND-NET communication

ND-NET is an optional part of SINTRAN providing communication between two or more independent, possibly geographically separated, ND computers.

A communication line is divided into a number of logical channels, up to 16 channels per line. The ND-NET system will route each channel from the sender to the receiver, making each logical connection independent of others using the same physical line.

Some of these channels are usually connected to background programs, analogous to the local background programs. A user at a remote terminal may log in as if he was working on a local machine. This is described in the SINTRAN Communication guide.

Channels without background programs are used for file transfer or for program-to-program communication. An RT program may reserve a channel as it reserves an internal device, and the operation is similar.

20.1.1 Reserving a channel

The logical number of the channel to be reserved must be known by the program, and the channel must be known to the program in the other end. However, the channel numbers are not necessarily the same in both ends of the connection. This is a system configuration parameter.

To reserve channel 617B for input,

```
CALL RESRV(617B, 0, 0)
```

Supposedly, another program on the remote computer has reserved the output side of the same channel, and will transmit data to this program.

20.1.2 Monitor calls permitted to operate on a channel

The channel permits sequential access only, primarily through the INBT/OUTBT calls. The READ and WRITE statements of Fortran, and their equivalent in other languages, translate to INBT/OUTBT and are thus permitted.

The multi-byte monitor calls: B8INB, M8INB, B4INW for input, B8OUT, M8OUT for output, are also permitted, and will reduce overhead somewhat.

MAGTP is allowed to operate on a channel, but the only function codes permitted are Read Record and Write Record. RFILE/WFILE are also permitted, but the block number parameter is ignored.

The receiving program, i.e. the output side of the channel, may define a break strategy through BRKM, MON 4. As long as no break condition occurs, the receiving program will not be restarted. A break condition will initiate the transfer of all characters since the last break and restart the program. Selecting a break strategy that gives as few breaks as possible significantly reduces communication overhead.

The IOSET call has a number of functions for clearing buffers, initiating transfer even though no break condition has occured, or setting break strategy. IOSET applied to ND-NET channels is documented in SINTRAN III Communication guide.

20.1.3 Clearing the buffer

If a program using the channel terminates abnormally, the buffers are not properly cleaned. After reserving a channel, the input buffer should therefore be cleared through the CIBUF call:

```
CALL CIBUF(617B)
```

The output buffer may be cleared by the sending program through the corresponding COBUF call.

20.1.4 Waiting for input request

By default, the communication on the channel is buffered on both sides. This may sometimes cause problems if the program system communicates with a terminal user: the output to the terminal may be sent before the user is able to handle it, echoing may be out of step with the input.

In order to synchronize the sender and the receiver, the sender may use the WRQI call to wait for an input request from the remote program. As soon as the buffers are empty and the receiver requests more input, the sender is restarted.

The call is available in background programs as well, and uses the background parameter mechanism: the T register contains the device number, normal return is to the second location following the call ("skip return"). Error return to the first location after the call will leave a file system error code in the A register. The call is not avilable in the Fortran library.

```
WRQI=163
QERMS=65

LDT (617
MON WRQI
MON QERMS
```

When the user presses the ESC key, a break condition will occur at the receiving side, restarting the communication program. The ESC character may be replaced by another character by setting the user break character through the MSDAE call, see section 13.14.3. This is particulary useful with terminals using ESC-sequences to signal various function codes.

MSDAE may also be used to redefine the character that brings user control back to the local computer, default value DEL. Redefining the disconnect character allows DEL to be used by editors and command processors to delete the last typed character.

20.1.5 File access

A file may be opened on a remote computer with an OPEN call. The channel to be used must have been defined as a peripheral file by the system supervisor through the @SET-PERIPHERAL-FILE command (not permitted for user RT).

The name thus given a channel must prefix the file name, separated by a dot. For example, if the channel is named CHA-617, and the file name on the remote computer is (RONNIE)BUDGET:DATA, the file may be opened through

```
OPEN(11, FILE='CHA-617.(RONNIE)BUDGET:DATA', ACCESS='R')
```

Analogously to the @RTENTER command on the local system, user SYSTEM must before remote file access is legal execute the command

@REMOTE-PASSWORD <line no.> <password>

The password of user RT on the remote machine is specified as the second parameter. This will allow RT programs using any of the channels on the specified <line no.> to use files.

20.1.6 **Allowed file operations**

The opened file is accessed as if it were an internal device; only sequential access is permitted and a number of monitor calls for disk files do not apply to communication channels.

These include RFILE/WFILE specifying a block number, SETBS, SETBL and SETBT etc. If the file is a remote terminal, the escape function may not be disabled.

The echo and break mode of a remote background terminal may be set by the program having reserved the input part. This is done exactly as for local terminals.

20.2 XMSG communication

The XMSG function is invoked through the XMSG monitor call, MON 200. This call is not available in the Fortran library, and an interface routine must be written to pick up the arguments from the Fortran argument list and load them into the proper registers.

An RT program is an example of an XMSG task. A task is an activity capable of transmitting and receiving messages. Each task using XMSG must have one or more ports, "mailboxes" named by alphanumeric strings. The name is used to find the receiver of a message; when communication is established a number returned by XMSG is used (this number is sometimes termed a magic number).

An XMSG communication session comprise several steps, each identified by the function code (XF...) to the XMSG monitor call:

- Obtaining a port through which messages may be sent and received (XFOPN). Several ports may be used, but messages to or from any other port can be sent or received through the same port. Sender and receiver have different ports, and the receiving port is identified when a message is sent.

- If communication is initiated from the port, obtaining a message buffer (XFGET). The buffer is used to contain the data being transferred, and may be reused for the next (or answer) message after data is read. In a simple dialoge, one buffer is needed and is sent back and forth between the participating tasks.

- Writing into the message buffer (XFWRI). User data are copied into the message, and may be done in several steps, with direct addressing within the buffer.

- Sending the message buffer to the receiver (XFSND). The first message in a dialoge contains the name of the receiver, and the message is sent to a standard task XROUT that will find the receiver in its tables. Later messages may be sent directly from the sender to the receiver.

- Receiving a message on a given port (XFRCV). The receiver acquires the first message sent to the port; if several have arrived they are queued and received one by one.

- Reading the data in the message (XFREA). Data are copied from the message into the user area. Reading does not destroy the data, and may be done with direct addressing within the buffer.

- Releasing a message buffer to the system (XFREL). If the buffer of a received message is not used for a new message, it should be released. The space will be released automatically when the port is closed.

20.2.1 Example: an RT service for background programs

A background program may interact with an RT program, for example through an internal device, and need to control the starting and stopping of the RT program. However, the RT and ABORT monitor calls are not allowed from background.

In order to provide those calls to background programs, an RT program is written to execute these calls. The background program will send a message to the port RTSERVICE, owned by the program SERVER (the name of the RT program is irrelevant to the background program). The message consists of a function code and the symbolic name of the program.

In order to find the RTSERVICE port, requests are sent through the XROUT name server. XROUT will interpret the message according to the function code in the first byte (starting at zero - byte 0 has the value 0) of the message. Function XSLET will forward the entire message to the named port.

The total length of the message is found in bytes 2 and 3.

XROUT recognizes the name RTSERVICE as a parameter. A byte of value -1 (377B) identifies parameter 1 as a string, and its length follows in the next byte.

The remainder of the message is ignored by XROUT, but the entire message is forwarded to RTSERVICE, which will find the function code in the 6th word (bytes 10:11), the RT name blank filled in words 7:9 (bytes 12:18).

The RT program will be in an infinite loop executing requests, and will not close the RTSERVICE port (the port will be automatically be closed if the program is aborted).

The background program may use the services by the function calls:

```
      PROGRAM BACKGR

C     Initial actions before starting a program

      CALL STARTPROG('PROG1')

C     Perform other actions (e.g. interact with PROG1 through
C     an internal device) before stopping it

      CALL STOPPROG('PROG1')
      END
```

The RT program SERVER will after initialization go into an infinite loop:

```
      PROGRAM SERVER, 40

      CALL INITPORT(0)
      DO WHILE(.TRUE.)
         CALL READFUNCTION(FNC, RTNAME)
         IF (FNC.EQ.RTCALL) CALL RT(RTNAME)
```

```
      IF (FNC.EQ.ABORTCALL) CALL ABORT(RTNAME)
ENDDO
END
```

The complete background program:

```
$CONSTANT STR1  = 177411B, XFGET =    2B, XFWRI = 7B
$CONSTANT XFSROU= 2014B,   XFOPN =   12B, XSLET = 101B

      SUBROUTINE SENDMESSAGE
      COMMON /MSG/STATUS, IMESSAGE, PORTNO
      INTEGER STATUS, IMESSAGE(15), PORTNO
      EXTERNAL IADDR
      CALL XMSG(XFGET, 30, 0, 0)
      CALL XMSG(XFWRI, IADDR(IMESSAGE), 30, 0)
      CALL XMSG(XFSROU, 0, 0, PORTNO)
      END

      SUBROUTINE MAKEMESSAGE(STR, FN)
      CHARACTER*(*) STR
      COMMON /MSG/STATUS, IMESSAGE, PORTNO
      INTEGER STATUS, IMESSAGE(15), PORTNO, FN, UPLIM
      CHARACTER*30 MESSAGE
      EQUIVALENCE (IMESSAGE,MESSAGE)
      IMESSAGE(1)   = XSLET
      IMESSAGE(2)   = 19 + LEN(STR)
      IMESSAGE(3)   = STR1
      MESSAGE(7:15) = 'RTSERVICE'
      IMESSAGE(9)   = FN
      UPLIM=18+LEN(STR)
      MESSAGE (19:UPLIM)= STR(1:UPLIM)
      MESSAGE (UPLIM+1:UPLIM+1)=''''
      END

      SUBROUTINE STARTPROG(STR)
      CHARACTER*(*) STR
      CALL MAKEMESSAGE(STR, 1)
      CALL SENDMESSAGE(0)
      END

      SUBROUTINE STOPPROG(STR)
      CHARACTER*(*)STR
      CALL MAKEMESSAGE(STR, 2)
      CALL SENDMESSAGE(0)
      END

      PROGRAM BACKGR
      EXTERNAL XMSG
      COMMON /MSG/STATUS, IMESSAGE, PORTNO
      INTEGER STATUS, IMESSAGE(15), PORTNO, XMSG
      PORTNO=XMSG(XFOPN, 0, 0, 0)
      CALL STARTPROG('MANE')
      CALL STOPPROG ('MANE')
      END
```

The RT program that performs the service:

```
$CONSTANT  XFOPN = 12B,   XFREA = 6B,      XFGET = 2B
$CONSTANT  XFSROU= 2014B, XFRCV = 100015B, XSNAM = 102B

       SUBROUTINE INITPORT
       EXTERNAL XMSG, IADDR
       COMMON /MSG/STATUS, IMESSAGE, PORTNO
       INTEGER STATUS, IMESSAGE(15), PORTNO, XMSG, IADDR
       CHARACTER*30 MESSAGE
       EQUIVALENCE (IMESSAGE,MESSAGE)

       PORTNO=XMSG(XFOPN, 0, 0, 0)
       CALL XMSG(XFGET,30,0,0)
       IMESSAGE(1)    =XSNAM
       IMESSAGE(2)    =11
       IMESSAGE(3)    =STR1
       MESSAGE(7:15) = 'RTSERVICE'
       CALL XMSG(XFWRI, IADDR(IMESSAGE), 30, 0)
       CALL XMSG(XFSROU, 0, 0, PORTNO)
       END

       SUBROUTINE READFUNCTION(FNC)
       COMMON /MSG/STATUS, IMESSAGE, PORTNO
       INTEGER STATUS, IMESSAGE(15), PORTNO, FNC
       EXTERNAL IADDR

       CALL XMSG(XFRCV, PORTNO, 0,0)
       CALL XMSG(XFREA, IADDR(IMESSAGE), 30, 0)
       FNC= IMESSAGE(9)
       END

       PROGRAM SERVER,30
       EXTERNAL GRTDA
       COMMON /MSG/STATUS, IMESSAGE, PORTNO
       INTEGER STATUS, IMESSAGE(15), PORTNO, GRTDA, RTADR, FNC

       CALL INITPORT
       DO WHILE (.TRUE.)
          CALL READFUNCTION(FNC)
          RTADR = GRTDA(IMESSAGE(10))
          IF (RTADR.NE.-1) THEN
             IF(FNC.EQ.1) CALL RT(RTADR)
             IF(FNC.EQ.2) CALL ABORT(RTADR)
          ENDIF
       ENDDO
       END
```

The interface to the XMSG monitor call is written as a Planc routine, using inline assembly code. The module also contains a routine to return the address of an integer as an integer value.

```
MODULE XM
EXPORT XMSG

TYPE COM = RECORD
       INTEGER ARRAY: IMESSAGE(1:15)
       INTEGER: PORTNO
ENDRECORD
IMPORT (COMMON) COM: MSG
INTEGER ARRAY: STACK(0:100)

% Interface to MON XMSG:

ROUTINE STANDARD VOID,INTEGER(INTEGER,INTEGER,INTEGER,INTEGER): &
        XMSG(T, A, D, X)
INTEGER: AR

INISTACK STACK
$*     LDX I X
$*     LDA I D
$*     COPY SA DD
$*     LDA I A
$*     LDT I T
$*     MON 200
$*     STA AR
       AR RETURN
ENDROUTINE

% Return the address of an integer:

ROUTINE STANDARD VOID,INTEGER POINTER(INTEGER): IADDR(VAR)

       INISTACK STACK
       ADDR(VAR) RETURN
ENDROUTINE

ENDMODULE
$EOF
```

APPENDIX A: Examples

1 A simple periodical RT program

This example shows the compilation, loading and execution of a simple RT program TIME, which is set up for periodical execution.

The program reserves the input part of the terminal from which the background work is done, so it will have to wait until the timesharing user RT has logged out. Then it reads one character from the terminal, and aborts itself if this is an "A".

If the character is not an "A", TIME will release the input part of the terminal, read the clock, reserve the output part of the terminal, write a "bell" character (which will cause an audible sound), a counting variable and the time and date on the terminal. Then it will release the output part of the terminal.

At the end of the program (line 21), TIME will enter a waiting state for an unspecified amount of time. When activated again, it will continue at the next statement (line 22) jumping to label 10, read the clock again etc.

However, every fourth time TIME will terminate itself. The next activation will cause execution to start from the beginning of the program.

TIME does not use recursion or reentrant routines, thus, the program is compiled without using the REENTRANT-MODE command to the FORTRAN compiler. It is loaded by the *LOAD command in the RT loader. (If the FTN compiler was used, the *NREENTRANT-LOAD command would have been preferred, and no explicit loading of the library would be necessary.)

```
@FORTRAN-100
ND-100 ANSI 77 FORTRAN COMPILER - NOVEMBER 24, 1981

FTN:COM EX-1,1,EX-1

ND-100 ANSI 77 FORTRAN COMPILER    13:43  27 NOV 1981
SOURCE FILE:  EX-1

    1*      C The program TIME reads a character from the
    2*      C terminal. If the character is A, the program aborts
    3*      C itself.  Otherwise, it writes a "bell", the time
    4*      C and the date on the terminal.
    5*      C
    6*            PROGRAM TIME,100
    7*            DIMENSION KLOKK(7)
    8*            IBELL=7*256
    9*            IDEV=52
   10*            CALL RESRV(IDEV,0,0)
   11*            IA=INCH(IDEV)
   12*            IF(IA.EQ.101B) CALL ABORT(0)
   13*            CALL RELES(IDEV,0)
   14*      10    CALL CLOCK(KLOKK)
   15*            CALL RESRV(IDEV,1,0)
   16*            WRITE(IDEV,100) IBELL,I,KLOKK
   17*            CALL RELES(IDEV,1)
   18*      100   FORMAT(/,1H+,A1,I5,'  TIME AND DATE IS :',7I5,/)
   19*            I=I+1
   20*            IF (I/4*4.EQ.I) CALL RTEXT
   21*            CALL RTWT
   22*            GOTO 10
   23*            END

- CPU TIME USED: 2.3 SECONDS.  23 LINES COMPILED.
- NO MESSAGES
- CODE SIZE=184 DATA SIZE=0 COMMON SIZE=0 STACK SIZE=18
FTN:EXIT
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NEW-SEGMENT 272,,,,
*PRESET-COMM 272 40000
*LOAD EX-1,,
*LOAD FORTRAN-1B-LIB,,
*END-LOAD
*EXIT
@PRIOR TIME 31

@CC Periodic execution every 5 seconds:

@INTV TIME 5 2
```

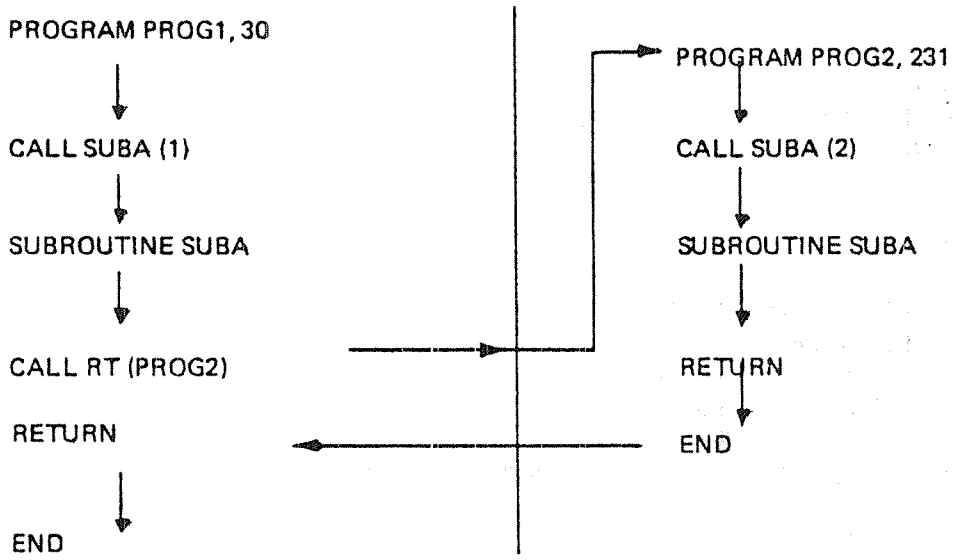
```
@CC Start execution:

@RT TIME

@LIST-RT-DESCRIPTION TIME
IN TIME QUEUE
INTV  RING:0 PRIORITY:     31
TIME LEFT:    4 SECS
INTERVAL:     5 SECS
START ADDRESS:     11, SEGMENTS:      0    272
P=      11
X=      67
T=       0
A=       0
D=       1
L=     110
S=       0
B=     424
READY
ACTUAL SEGM.:      0    272
@LIST-TIME-QUEUE
RTSLI
TIMRT
TIME

@LIST-RT-DESCRIPTION TIME
IN TIME QUEUE
INTV  RING:0 PRIORITY:     31
TIME LEFT:    4 SECS
INTERVAL:     5 SECS
START ADDRESS:     11, SEGMENTS:      0    272
P=      11
X=      67
T=       0
A=       0
D=       1
L=     110
S=       0
B=     424
READY
ACTUAL SEGM.:      0    272
@LOG

B

 0  TIME AND DATE IS :   40   44   38   14   27   11   1981

 1  TIME AND DATE IS :   23   44   38   14   27   11   1981

 2  TIME AND DATE IS :   40   53   38   14   27   11   1981

 3  TIME AND DATE IS :   40   58   38   14   27   11   1981

A
```

2 Two RT programs calling a reentrant subroutine

The following example shows the compilation, loading and execution of two RT programs PROG1 and PROG2 calling the same subroutine SUBA.

SUBA has one parameter, IP. When IP=1, SUBA will activate PROG2. IP will be written onto the terminal by SUBA before and after the test on IP, thus demonstrating how the CPU control changes between the two RT programs.

PROG1 calls SUBA will the parameter IP=1 which leads to an activation of PROG2. Because PROG2 has a higher priority than PROG1, this will result in the following CPU control flow:



Note that SUBA is really executed twice simultaneously. This is confirmed by the order in which the output from SUBA arrives at the terminal.

Note also that if the priority of PROG2 had been equal to or lower than the priority of PROG1, PROG1 would have finished completely before PROGR2 would have been started (assuming that all required resources were available when reservation was attempted).

The subroutine SUBA must be reentrant, because it is used by both PROG1 and PROG2 simultaneously. Accordingly, all programs calling it must also be reentrant. In order to produce reentrant code, the command REENTRANT-MODE must be given to the FORTRAN compiler prior to compilation of PROG1, SUB1 and PROG2. (Note that a reentrant routine may call a non-reentrant one, but a reentrant routine may only be called by reentrant routines).

If the FTN compiler is used, the RT loader command *REENTRANT-LOAD should be used to load the program. This will allocate a stack and load the reentrant Fortran library automatically.

By default, the segment will be a non-demand segment so that the entire segment will be transferred to main memory before execution of PROG1 starts, and there will be no additional swapping before PROG2 starts.

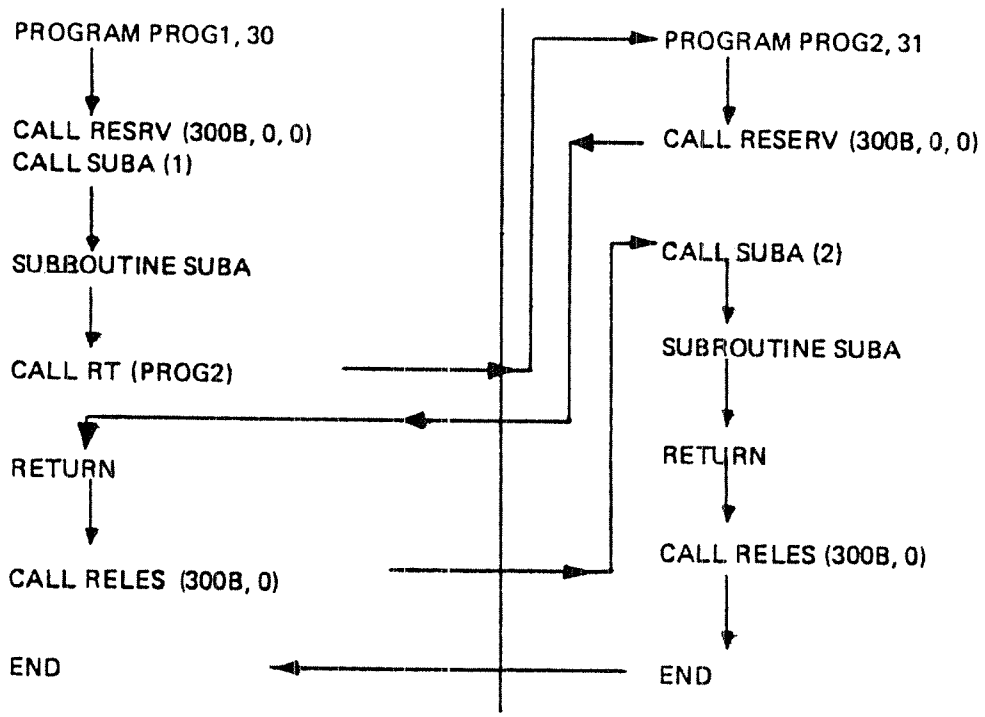
```
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$REENTRANT-MODE
$COMPILE EX-2-P1,1,P1
    1*              PROGRAM PROG1,30
    2*              CALL SUBA(1)
    3*              END
    4*
    5*              SUBROUTINE SUBA(IP)
    6*              EXTERNAL PROG2
    7*              NO=52
    8*              CALL RESRV(NO,1,0)
    9*              WRITE(NO,100)IP
   10*              CALL RELES(NO,1)
   11*              IF (IP.EQ.1) CALL RT(PROG2)
   12*              CALL RESRV(NO,1,0)
   13*              WRITE(NO,100)IP
   14*              CALL RELES(NO,1)
   15*  100         FORMAT(1X,I2)
   16*              END

16 LINES COMPILED . OCTAL SIZE=     115
CPU-TIME USED IS 0.4 SEC.
$COMPILE EX-2-P2,1,P2
    1*              PROGRAM PROG2,31
    2*              CALL SUBA(2)
    3*              END

3 LINES COMPILED . OCTAL SIZE=      14
CPU-TIME USED IS 0.1 SEC.
$EXIT
```

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*REENTRANT-LOAD P1,,
NEW SEGMENT NO: 242
STACK LENGTH:
*REENTRANT-LOAD p2,,
STACK LENGTH:
*END-LOAD
*WRITE-RTFIL
SEGMENT NO: 242
OUTPUT FILE:

 PROG1     40235  242     0
 SUBA         13  242
 8LEAV     10607  242
 RESRV       115  242
 8FIO        443  242
 RELES       117  242
 RT          113  242
 PROG2     40267  242     0
 8OFIL       440  242
 8OVTB     10631  242
 8RUTB     10633  242
 8OVNO       441  242
 8ALTF     10634  242
 8DAD      10171  242
 8DSB      10173  242
 8DMU      10434  242
 8RLDN     10167  242
 OUTBT     10640  242
 INBT      10635  242
 8ERR       6541  242
*EXIT

@RT PROG1
@LOG
 20.57.16       29 DECEMBER 1981
 TIME USED IS     8 SECONDS OUT OF    14 MINUTES 26 SECONDS

  1
  2
  2
  1
```

3 Two RT programs calling a nonreentrant subroutine protected by a semaphore

In this example, the program system in the previous example is compiled without giving the command REENTRANT-MODE. To prevent simultaneous execution of SUBA, it is protected by semaphore 300B.

Consider the illustration below, where the RT programs PROG1 and PROG2 both reserve semaphore 300B before they call SUBA. In this case, the CPU control flow will be as follows:



PROG1 is interrupted by PROG2 with a higher priority, but PROG2 will have to wait for semaphore 300B until PROG1 releases it. The order by which SUBA is executed is illustrated by the output on the terminal.

Here we see even more clearly the general principle that the RT program with the highest priority and something to do will always get the CPU first.

PROG1, SUBA and PROG2 may now be compiled without using the command REENTRANT-MODE, and loaded using the *NREENTRANT-LOAD (if the FTN compiler is used) or *LOAD (if FORTRAN-100 is used). No allocation of stack is necessary.

If the programs PROG1 and PROG2 in the previous example are still in the system, they should be deleted and the segment cleared before the new programs with the same names are loaded.

All programs and subprograms are loaded on the same segment which, by default, wil be a non-demand segment. Thus, there is no swapping of pages during execution.

```
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$COMPILE EX-3-P1,1,P1
    1*          PROGRAM PROG1,30
    2*          CALL RESRV(300B,0,0)
    3*          CALL SUBA(1)
    4*          CALL RELES(300B,0)
    5*          END
    6*
    7*          SUBROUTINE SUBA(IP)
    8*          EXTERNAL PROG2
    9*          NO=52
   10*          CALL RESRV(NO,1,0)
   11*          WRITE(NO,100)IP
   12*          CALL RELES(NO,1)
   13*          IF(IP.EQ.1) CALL RT(PROG2)
   14*          CALL RESRV(NO,1,0)
   15*          WRITE(NO,100)IP
   16*          CALL RELES(NO,1)
   17*  100     FORMAT (1X,I2)
   18*          END

18 LINES COMPILED . OCTAL SIZE=    236
CPU-TIME USED IS 0.5 SEC.
$COMPILE EX-3-P2,1,P2
    1*            PROGRAM PROG2,31
    2*            CALL RESRV(300B,0,0)
    3*            CALL SUBA(2)
    4*            CALL RELES(300B,0)
    5*            END

5 LINES COMPILED . OCTAL SIZE=     72
CPU-TIME USED IS 0.1 SEC.
$EXIT
```

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NEW-SEGMENT 270,,,,
*NREENTRANT-LOAD P1,,
*NREENTRANT-LOAD P2,,
*END-LOAD
*WRITE-RTFIL
SEGMENT NO: 270
OUTPUT FILE:

 PROG1    40235  270    0
 8RTEN      334  270
 RESRV      327  270
 SUBA        71  270
 RELES      331  270
 8LEAV    12657  270
 8ENTR      403  270
 8FIO       614  270
 RT         325  270
 PROG2    40267  270    0
 8OFIL      611  270
 8OVTB    12676  270
 8RUTB    12700  270
 8OVNO      612  270
 8ALTF    12703  270
 8CONV    12622  270
 8RANT    11773  270
 8DAD     11211  270
 8DSB     11213  270
 8DMU     11454  270
 8RLDN    11207  270
 OUTBT    12504  270
 8BINI    12354  270
 8BINO    12527  270
 INBT     12174  270
 8ERR      7561  270
 8CNCT    11627  270
 8CRAN    12636  270
 8CLSB    12701  270

*EXIT

@RT PROG1
@LOG
 21.17.16      29 DECEMBER 1981
 TIME USED IS    9 SECONDS OUT OF   10 MINUTES 3 SECONDS

  1
  1
  2
  2
```

4 RT programs using two segments

This example demonstrates how the program system in the previous example could utilize two segments. The loading and execution is shown.

PROG1, SUBA, and the necessary routines from FORTRAN-LIB are loaded to segment 273. PROG2 is loaded to segment 274. The segments are built one by one.

Now the problem arises that if segment 273 is built first, PROG2 which is activated from SUBA will be undefined when closing the segment. On the other hand, if segment 107 is established first, SUBA and the library routines which are called from PROG2 will be undefined when closing this segment.

One way of solving the problem is illustrated.

Segment 273 is loaded with PROG1 and SUBA. The undefined symbol PROG2 is declared as the name of an RT program to be defined later, and then the segment is closed.

Segment 274 is then loaded with PROG2, and segment 273 is declared as the link segment. This makes all defined symbols on segment 273 known before closing segment 274. The routines already loaded to segment 273 will thus not need to be loaded to segment 274, but appropriate links to segment 273 will be established by the loader.

Whe executing PROG2, the two segments constitute the logical address space, and now a new problem occurs: segments 273 and 274 may not overlap in logical address space, since both segments are used when executing PROG2. The RT loader checks this when a link segment has been defined, and gives an error message if address overlap occurs.

The first logical page number of segment 274 must be higher than the last logical page number of segment 273. By using the command *WRITE-LOAD-ADDRESS 273 we will get information about the lower and upper load addresses used by segment 273.

The lower address of segment 274 must be higher than the upper load address of segment 273, rounded up to the next page limit. In other words, the lowest address on segment 274 is the first address that is evenly divisible by 2000B above the uppermost load address on segment 273.

The *SET-LOAD-ADDRESS command should be before any code is loaded to the segment (it may be used later, but will then apply only to the code loaded after the command is issued), and it can only be used on a segment being built. Therefore, segment 274 should be declared by a *NEW-SEGMENT command and the load address set, rather than automatically allocating a segment at the first *NREENTRANT-LOAD. (If FORTRAN-100 rather than FTN is used, the most appropriate command is *LOAD, which will not in any case allocate a new segment.)

After loading, PROG1 is set up for periodical execution every 5 seconds and started. After that, the user logs out.

The surprising output shows that PROG1 is executed to completion twice, succeeded by two executions of PROG2. Later executions cause a change of CPU control corresponding to the discussion of the previous example. The situation with two succeeding executions of PROG1 followed by two edections of PROG2 might occur again at any time.

How is this possible even when PROG2 has higher priority than PROG1?

The explanation is that segment 273 will be swapped to memory when the command @RT PROG1 is issued. Segmnent 274 will not be swapped in before PROG2 is activated from PROG1 in SUBA. PROG2 will, however, not be started before all pages belonging to segment 274 have been loaded. While waiting for the disk transfer the next RT program in the executionqueue, PROG1, will continue execution.

Because of th time used for the logging out procedure and the short interval of PROG1, this program will be set up for repetition and executed twice before PROG2 gets the pages of segment 274 into memory.

The execution of RT programs may thus be greatly influenced by the size of physical memory and the other activity in the computer.

As shown in the example the order of execution of the two programs after having fixed segment 274 will be the same as in example 3.

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

#NEW-SEGMENT 273,,,,
#NREENTRANT-LOAD P1
LINKING-SEGMENT NO.:
#DECLARE-PROGRAM PROG2
RT-DESCRIPTION ADDRESS:
#WRITE-LOAD-ADDRESS
SEGMENT NO:

L.ADR:       0  U.ADR:     233  C.LADR:     234
#END-LOAD
#WRITE-SEGMENT
SEGMENT NO: 273
OUTPUT FILE:

 273       0  13777  1465  0  0  1 RFW NON DEMAND
#NEW-SEGMENT
SEGMENT NO: 274
RING:
SEGMENT TYPE:
PROTECTION BITS:
WP/NP:
#SET-LOAD-ADDRESS
SEGMENT NO:
ADDRESS: 14000
#NREENTRANT-LOAD P2,273
#END
#EXIT
@INTV PROG1 5 2
@RT PROG1
@LOG
 14.59.17        30 DECEMBER    1981
 TIME USED IS  21 SECONDS OUT OF    16 MINUTES    11 SECONDS
--EXIT--

  1
  1
  1
  1
  2
  2
  2
  2
  1
  1
  2
  2
  1
  1
```

```
 15.00.04        30 DECEMBER  1981
 SINTRAN III - VS    VERSION G
ENTER RT
PASSWORD:
OK
PROJECT PASSWORD:

PROJECT NAME: REAL-TIME-GUIDE
@ABORT PROG1
@ABORT PROG2
@FIX 274
@LIST-SEGMENT 274

FIRST PAGE:     106  LENGTH:        1
SEG.FILE:       0 MASS. ADR:     1017
WPM RPM FPM FIX OK

@INTV PROG1 5 2
@RT PROG1
@LOG
 15.08.10        30 DECEMBER   1981
 TIME USED IS   4 SECONDS OUT OF   8 MINUTES   7 SECONDS
--EXIT--

  1
  1
  2
  2
  1
  1
  2
  2
  1
 15.09.00        30 DECEMBER  1981
 SINTRAN III - VS    VERSION G
ENTER RT
PASSWORD:
OK
PROJECT PASSWORD:

PROJECT NAME: REAL-TIME-GUIDE
@ABORT PROG1
@ABORT PROG2
@UNFIX 274
@LOG
 15.08.40        30 DECEMBER   1981
 TIME USED IS   0 SECONDS OUT OF    40 SECONDS
--EXIT--
```

Now, an alternative way of loading the same programs to two segments will be shown.

It is possible to build both segments concurrently, in which case the loader will take care of load addresses so that no overlap occurs. Default load address for the two segments are 0 and 100000B, respectively.

Again, PROG1, SUBA and library routines are loaded to one segment and PROG2 to another segment.

The segments 273 and 274 are explicitly allocated by the *NEW-SEGMENT commands, before PROG1 and SUBA are loaded to the first of these by *NREENTRANT-LOAD.

In order to load PROG2 to segement 274, *LOAD command must be used. This implies that FTNLIBR will not be automatically loaded when the *END-LOAD command is given; automatic loading occurs only if *NREENTRANT-LOAD was the last load command given. Thus, loading of the necessary library routines to segment 273 is done by an explicit *LOAD command.

*END-LOAD will close both segments. The result of this loading procedure is the same as that of the previous one, except for the logical addresses on segment 274.

The *WRITE-SEGMENT commands show the extent of the area on the segements to which code was loaded.

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NEW-SEGMENT 273,,,,
*NEW-SEGMENT 274,,,,
*NREENTRANT-LOAD  P1,,
*LOAD FTNLIBR
LOAD-SEGMENT NO.: 273
LINKING-SEGMENT NO.:
*LOAD P2, 274, 273
*WRITE-LOAD-ADDRESS 273

L.ADR:      0   U.ADR:   11276   C.LADR:   11277
*WRITE-LOAD-ADDRESS 274

L.ADR:  100000  U.ADR:  100070   C.LADR:  100071
*END-LOAD
*WRITE-SEGMENTS
SEGMENT NO: 273
OUTPUT FILE:

 273      0   11777   3005   0   0   1 RWF NON DEMAND
*WRITE-SEGMENTS 274,,

 274 100000  101777   2750   0   0   1 RWF NON DEMAND
*EXIT
@INTV PROG1 5 2
@RT PROG1
@LOG
 15.38.00        30 DECEMBER   1981
 TIME USED IS  23 SECONDS OUT OF    58 SECONDS
--EXIT--
  1
  1
  1
  1
  2
  2
  2
  2
  1
  1
  2
  2
  1
  1
```

5 A recursive function

This example shows the classical example of calculating N factorial:

```
N! = N * (n-1) * ... * 2 * 1
```

by means of a function calling itself

Recursive functions and subroutines are normally not allowed in Fortran, because of certain pecularities in the standard Fortran syntax. In FTN and FORTRAN-100, recursion is allowed provided the program is compiled in reentrant mode. Such programs are not strictly in accordance with the Fortran standard.

The program uses double precision integer values to allow values of N up to 12.

Since the compilation and loading was done from terminal 52 the command @LIST-DEVICE 52 shows that the background program has reserved the input and output parts while the RT program NFAC is in the waiting queue for the input part of this terminal

```
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$REENTRANT-MODE
$PROGRAM MAP
$COMPILE EX-5--NFAC,1,NFAC
    1*  C Example of a recursive function, FACU, calling itself.
    2*  C The main program NFAC reserves the input part of terminal
    3*  C number 52, reads the number N for which the factorial
    4*  C is to be calculated and releases the terminal. It
    5*  C then calls the recursive function, FACU, which performs
    6*  C each multiplication by calling itself with the parameter
    7*  C decreased by 1. After the multiplication it reserves
    8*  C the output part of the terminal, outputs the intermediate
    9*  C result, releases the terminal and returns to the caller
   10*
   11*            PROGRAM NFAC,30
   12*            COMMON IDEV
   13*            DOUBLE INTEGER NF,FACU
   14*            IDEV=52
   15*            CALL RESRV(IDEV,0,0)
   16*            INPUT(IDEV)N
   17*            CALL RELES(IDEV,0)
   18*            NF=FACU(N)
   19*            CALL RESRV(IDEV,1,0)
   20*            WRITE(IDEV,100) N,NF
   21*  100       FORMAT(I3,'! =',I14)
   22*            CALL RELES(IDEV,1)
   23*            END

---------- M E M O R Y  A D D R E S S  M A P ----------
 LINE:      +0    +1    +2    +3    +4    +5    +6    +7    +8    +9
   10*            0                       3     5    13    24    31    36
   20*      44          67    74

---------- L O C A L  I D E N T I F I E R S ----------

   DB INT. VARIABLE      NF   -   162
   INTEGER VARIABLE       N   -   163
            TOTAL NO OF STACK LOCATIONS     43 (OCTAL)

---------- C O M M O N  I D E N T I F I E R S ----------

   INTEGER VARIABLE    IDEV     BLANK        0

---------- E X T E R N A L  R E F E R E N C E S ----------

   DB INT.    FACU
    REAL     RESRV
    REAL      8FIO
    REAL     RELES

23 LINES COMPILED . OCTAL SIZE=     103
CPU-TIME USED IS 0.8 SEC.
```

```
$COMPILE EX-5-FACU,1,FACU
    1*              INTEGER FUNCTION FACU (K)
    2*              COMMON IDEV
    3*              DOUBLE INTEGER FACU,KK
    4*
    5*              IF (K.GT.0) THEN
    6*                    KK=K*FACU(K-1)
    7*                    CALL RESRV(IDEV,1,0)
    8*                    WRITE(IDEV,100) K,KK
    9*  100               FORMAT(I3,I14)
   10*                    CALL RELES(IDEV,1)
   11*                    FACU=KK
   12*              ELSE
   13*                    FACU=1
   14*              ENDIF
   15*              END

---------- M E M O R Y  A D D R E S S  M A P ----------
 LINE:      +0    +1    +2    +3    +4    +5    +6    +7    +8    +9
    0*          102                      104   110   127   135
   10*      155   162   165   165   170   170

---------- L O C A L  I D E N T I F I E R S ----------

   DB INT. VARIABLE    FACU   -   155
   INTEGER VARIABLE       K   PARAMETER
   DB INT. VARIABLE      KK   -   157
            TOTAL NO OF STACK LOCATIONS     50 (OCTAL)

---------- C O M M O N  I D E N T I F I E R S ----------

   INTEGER VARIABLE    IDEV     BLANK        0

---------- E X T E R N A L  R E F E R E N C E S ----------

    REAL     RESRV
    REAL      8FIO
    REAL     RELES

15 LINES COMPILED . OCTAL SIZE=     100
CPU-TIME USED IS 0.7 SEC.
$EXIT
```

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*REENTRANT-LOAD NFAC
LINKING-SEGMENT NO.:
STACK LENGTH:
NEW SEGMENT NO: 244
*REENTRANT-LOAD FACU,,,
*END
*EXIT

@RT NFAC
@LIST-RT-DESCRIPTION NFAC
 RING:0 PRIORITY:     30
LAST STARTED:     2 SECS
START ADDRESS:       0, SEGMENTS:       0     242
P=     103
X=   10727
T=       3
A=       0
D=     102
L=      13
S=     200
B=   11107
WAITING FOR:  23126
ACTUAL SEGM.:       0     242
@LIST-DEVICE
LOG. UNIT: 52
INOUT/OUTPUT(0 OR 1): 1
RESERVED BY: BAK13
@LOG
 15.42.18       30 DECEMBER 1981
 TIME USED IS    58  SECONDS OUT OF     21 MINUTES    14 SECONDS
--EXIT--
9
 1               1
 2               2
 3               6
 4              24
 5             120
 6             720
 7            5040
 8           40320
 9          362880
 9! =       362880
```

6 An RT program using three segments

The main program in this example calls the two subroutines SUBR1 and SUBR2. The RT program system is not reentrant.

The main program is loaded to segment 314 with the routines from FTNLIBR. SUBR1 and SUBR2 are loaded to segments 315 and 316 respectively so that the program system will utilize 3 segments.

During execution a problem will arise because it is not possible for a program to use more than two segments concurrently. Segment 314 will be readied for execution when PROG1 is started. Additionally, when SUBR1 is called segment 315 will be used. This segment must be exchanged with segment 316 by means of the monitor call MEXIT before SUBR2 is called.

Similarly, during loading it is not allowed to establish more than two segments simultaneously. Thus, segments 314 and 315 are loaded in parallel, both segments using the other as a link segment. (PROG1 will call SUBR1 on segment 315 and SUBR2 will call the library routines on segment 314.)

No logical address overlapping will take place between these two segments, because the default initial load address is 0 and 100000B respectively when two segments are built in parallel.

The symbol SUBR2 is not loaded to any of these two segments, but is referenced on segment 314, so it is necessary todefine an address corresponding to the symbol. When this is done the symbol must be deleted from the linking table before the segments are closed. Otherwise the error "DOUBLE DEFINITION" will occur when loading SUBR2 to segment 316 using 314 as a link segment. The linking to segment 314 saves space because the library routines are already present on segment 314. (The library routines could also have been duplicated on segment 316. This would make segment 316 self contained and usable with other segments than 314 as well.)

However, SUBR2 must be loaded to the address corresponding to the one define, so the *SET-LOAD-ADDRESS command must specify exactly the same address as given in the *DEFINE-SYMBOL command. This address must be outside the logical address space occupied by segment 314 to avoid overlapping. SUBR2 must be the first routine loaded after the *SET-LOAD-ADDRESS command is issued.

```
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$COMPILE EX-6-P1,1,P1
     1*           PROGRAM PROG1,31
     2*           COMMON ID,TERM
     3*           INTEGER TERM
     4*           ID=0
     5*           TERM=52
     6*           CALL RESRV(TERM,1,0)
     7*           WRITE(TERM,101)ID
     8*  101      FORMAT(' THIS IS PROG1 ',I5)
     9*           CALL SUBR1
    10*           WRITE(TERM,101)ID
    11*           CALL RESRV(TERM,0,0)
    12*           INPUT(TERM)NSEG
    13*           CALL RELES(TERM,0)
    14*           CALL MEXIT(NSEG)
    15*           CALL SUBR2
    16*           WRITE(TERM,101)ID
    17*           CALL RELES(TERM,1)
    18*           END

18 LINES COMPILED . OCTAL SIZE=     202
CPU-TIME USED IS 0.5 SEC.
$COMPILE EX-6-S1,1,S1
     1*           SUBROUTINE SUBR1
     2*           COMMON ID,TERM
     3*           INTEGER TERM
     4*           ID=ID+1
     5*           WRITE(TERM,101)ID
     6*  101      FORMAT(' THIS IS SUBR1 ',I5)
     7*           END

7 LINES COMPILED . OCTAL SIZE=      57
CPU-TIME USED IS 0.2 SEC.
$COMPILE EX-6-S2,1,S2
     1*           SUBROUTINE SUBR2
     2*           COMMON ID,TERM
     3*           INTEGER TERM
     4*           ID=ID+1
     5*           WRITE(TERM,101)ID
     6*  101      FORMAT(' THIS IS SUBR2 ',I5)
     7*           END

7 LINES COMPILED . OCTAL SIZE=      57
CPU-TIME USED IS 0.2 SEC.
$EXIT
```

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*NEW-SEGMENT 314,,,,
*NEW-SEGMENT 315,,,,
*LOAD P1,314,315
*LOAD FTNLIBR,314,315
*LOAD S1,315,314
*WRITE-REFERENCES,,

 SUBR2

*WRITE-LOAD-ADDRESS 314,,

L.ADR:       0  U.ADR:  12566  C.LADR:  12567
*DEFINE-SYMBOL SUBR2,14000,314
*WRITE-SYMBOLS,,

 8CLSB     12564 314
 8CRAN     12521 314
 8CNCT     11512 314
 8ERR       7444 314
 INBT      12057 314
 8BINO     12412 314
 8BINI     12237 314
 OUTBT     12367 314
 8RLDN     11072 314
 8DMU      11337 314
 8DSB      11076 314
 8DAD      11074 314
 8RANT     11656 314
 8CONV     12505 314
 8ALTF     12566 314
 8OVNO       475 314
 8RUTB     12563 314
 8OVTB     12561 314
 8OFIL       474 314
 8ENTR       266 314
 8LEAV     12542 314
 SUBR2     14000 314
 MEXIT       201 314
 RELES       214 314
 SUBR1    100000 315
 8FIO        477 314
 RESRV       212 314
 8RTEN       217 314

*DELETE-SYMBOL SUBR2
*END-LOAD
*WRITE-SEGMENTS 314,,

 314       0  13777   1571   0   0   1 RFW NON DEMAND
*WRITE-SEGMENTS 315,,

 315  100000 101777   1051   0   0   1 RFW NON DEMAND
```

```
*NEW-SEGMENT 316,,,,
*SET-LOAD-ADDRESS 316 14000
*NREENTRANT-LOAD (RTG)S2,314
*END-LOAD
*WRITE-RTFIL 314,,

 BLANK    12567  314
 PROG1    40353  314  315
 8RTEN      217  314
 RESRV      212  314
 8FIO       477  314
 RELES      214  314
 MEXIT      201  314
 8LEAV    12542  314
 8ENTR      266  314
 8OFIL      474  314
 8OVTB    12561  314
 8RUTB    12563  314
 8OVNO      475  314
 8ALTF    12566  314
 8CONV    12505  314
 8RANT    11656  314
 8DAD     11074  314
 8DSB     11076  314
 8DMU     11337  314
 8RLDN    11072  314
 OUTBT    12367  314
 8BINI    12237  314
 8BINO    12412  314
 INBT     12057  314
 8ERR      7444  314
 8CNCT    11512  314
 8CRAN    12521  314
 8CLSB    12564  314
*WRITE-RTFIL 315,,

 PROG1    40353  314  315
 SUBR1   100000  315

*WRITE-RTFIL 316,,

 SUBR2    14000  316

*EXIT
@RT PROG1
@LOG
 15.57.48       30 DECEMBER 1981
 TIME USED IS    30  SECONDS OUT OF     4 MINUTES    16 SECONDS
--EXIT--

THIS IS PROG1       0
THIS IS SUBR1       1
THIS IS PROG1       1
206
THIS IS SUBR2       2
THIS IS PROG1       2
```

Now, an alternative way of solving the problem of utilizing three segments will be shown.

The call for SUBR2 in PROG1 is replaced by a call for SUBR1. Then it is not necessary to define and delete the symbol on segment 314. On the other hand, the programmer must load SUBR2 on segment 316 to exactly the same address as the address of SUBR1 on segment 315.

The decimal segment number 206 given as input to PROG1 corresponds to the octal segment number 316B.

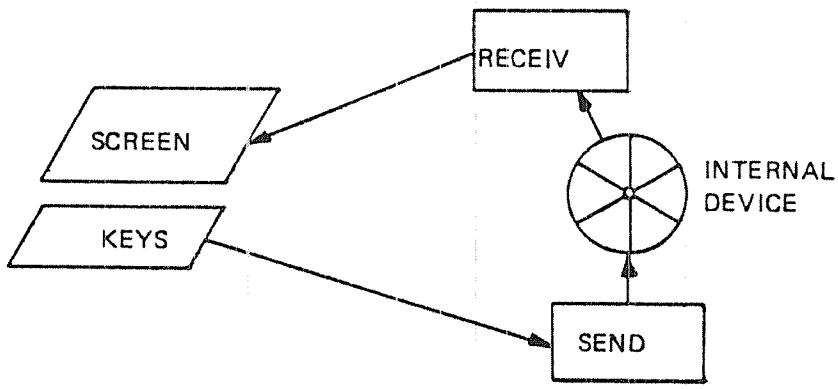
```
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$COMPILE EX-6-P1,1,P1
    1*          PROGRAM PROG1,31
    2*          COMMON ID,TERM
    3*          INTEGER TERM
    4*          ID=0
    5*          TERM=52
    6*          CALL RESRV(TERM,1,0)
    7*          WRITE(TERM,101)ID
    8*  101     FORMAT(' THIS IS PROG1 ',I5)
    9*          CALL SUBR1
   10*          WRITE(TERM,101)ID
   11*          CALL RESRV(TERM,0,0)
   12*          INPUT(TERM)NSEG
   13*          CALL RELES(TERM,0)
   14*          CALL MEXIT(NSEG)
   15*          CALL SUBR1
   16*          WRITE(TERM,101)ID
   17*          CALL RELES(TERM,1)
   18*          END

18 LINES COMPILED . OCTAL SIZE=     202
CPU-TIME USED IS 0.5 SEC.
$EXIT
```

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*CLEAR-SEGMENT 314

RT-PROGRAMS ON SEGMENT:

 PROG1

DELETING THIS RT-PROGRAM(S)? Y
*CLEAR-SEGMENT 315
*CLEAR-SEGMENT 316
*NEW-SEGMENT 314,,,,
*NEW-SEGMENT 315,,,,
*LOAD (RTG)P1,314,315
*LOAD FTNLIBR,314,315
*WRITE-REFERENCES,,

 SUBR1

*SET-LOAD-ADDRESS 315 14000
*LOAD (RTG)S1,315,314
*END-LOAD
*NEW-SEGMENT 316,,,,
*SET-LOAD-ADDRESS 316 14000
*NREENTRANT-LOAD (RTG)S2,314
*END-LOAD
*WRITE-RTFIL 314,,

 BLANK    12567  314
 PROG1    40353  314  315
 8RTEN      217  314
 RESRV      212  314
 8FIO       477  314
 RELES      214  314
 MEXIT      201  314
 8LEAV    12542  314
 8ENTR      266  314
 8OFIL      474  314
 8OVTB    12561  314
 8OVNO      475  314
 8ALTF    12566  314
 8CONV    12505  314
 8RANT    11656  314
 8DSB     11076  314
 8DMU     11337  314
 8RLDN    11072  314
 OUTBT    12367  314
 8BINI    12237  314
 8BINO    12412  314
 INBT     12057  314
 8ERR      7444  314
 8CNCT    11512  314
 8CRAN    12521  314
 8CLSB    12564  314
```

```
*WRITE-RTFIL 315,,

 PROG1    40353  314  315
 SUBR1    14000  315

*WRITE-RTFIL 316,,

 SUBR2    14000  316

*WRITE-SEGMENT 314,,

 314      0  13777  1571  0  0  1 RFW NON DEMAND
*WRITE-SEGMENT 315,,

 315  14000  15777  1051  0  0  1 RFW NON DEMAND
*WRITE-SEGMENT 316,,

 316  14000  15777  1052  0  0  1 RFW NON DEMAND
*EXIT
@RT PROG1
@LOG
 15.57.48       30 DECEMBER 1981
 TIME USED IS   30  SECONDS OUT OF    4 MINUTES   16 SECONDS
--EXIT--

THIS IS PROG1      0
THIS IS SUBR1      1
THIS IS PROG1      1
206
THIS IS SUBR2      2
THIS IS PROG1      2
```

7 Internal device

In this chapter an application of a byte oriented internal device is demonstrated. The RT program RECIEV reads characters from the internal device and outputs them to the terminal. THe RT program SEND writes the characters to the internal device, characters which have been input from the terminal. The effect will be a "double echo" of characters.



The above transport goes on until SEND reads the ASCII character 300B, "@" with even parity.

When starting the two programs (RECEIV is started from SEND) and listing their RT description, we find that SEND is in a waiting queue for the input part of the terminal, while RECEIV is in IOWAIT, waiting for input from the internal device.

First shown is compiling in reentrant mode, loading both programs to the same segment. Then is shown non-reentrant compilation and loading to two separate segments. Even the reentrant programs could have been loaded to separate segments, but using only one segment reduces overhead and wasted space.

```
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$REENTRANT-MODE
$COMPILE EX-7-S,1,S
    1*          PROGRAM SEND,10
    2*          EXTERNAL RECEIV
    3*          CALL RT(RECEIV)
    4*  10      CALL RESRV(52,0,0)
    5*          ICH=INCH(52)
    6*          IF (ICH.EQ.300B) CALL ABORT(0)
    7*          CALL RELES(52,0)
    8*          CALL RESRV(200B,1,0)
    9*          CALL OUTCH(200B,ICH)
   10*          CALL RELES(200B,1)
   11*          GOTO 10
   12*          END

12 LINES COMPILED . OCTAL SIZE=     72
CPU-TIME USED IS 0.3 SEC.
$COMPILE EX-7-R,1,R
    1*          PROGRAM RECEIV,20
    2*  10      CALL RESRV(200B,0,0)
    3*          ICH=INCH(200B)
    4*          CALL RELES(200B,0)
    5*          CALL RESRV(52,1,0)
    6*          CALL OUTCH(52,ICH)
    7*          CALL RELES(52,1)
    8*          GOTO 10
    9*          END

9 LINES COMPILED . OCTAL SIZE=     55
CPU-TIME USED IS 0.2 SEC.
$EXIT
```

```
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*REENTRANT-LOAD S,,
NEW SEGMENT NO: 244
*REENTRANT-LOAD R,,
*END-LOAD
*EXIT
@RT SEND
@LIST-EXECUTION-QUEUE
RTERR
BAK10
BAK13
RECEIV
BCH01
BAK35
DUMMY
@LIST-RT-DESCRIPTION SEND
 RING:0 PRIORITY:     10
LAST STARTED:     5 SECS
START ADDRESS:       0, SEGMENTS:       0    244
P=     124
X=     573
T=       3
A=       0
D=     123
L=      15
S=       0
B=     755
WAITING FOR:  23126
ACTUAL SEGM.:       0    244
@LIST-RT-DESCRIPTION RECEIV
I/O-WAIT  RING:0 PRIORITY:     20
LAST STARTED:     5 SECS
START ADDRESS:    2536, SEGMENTS:       0    244
P=      74
X=    3230
T=     200
A=    3230
D=      71
L=    2553
S=       0
B=    3410
READY
ACTUAL SEGM.:       0    244
RESERVED DATAFIELDS:
  33727
```

```
@LOG
 16.16.49       30 DECEMBER 1981
 TIME USED IS    12  SECONDS OUT OF     5 MINUTES    11 SECONDS
--EXIT--

AASSDDFFGGHHJJKKLLNN@

 16.17.14       30 DECEMBER 1981
 SINTRAN III - VS     VERSION G
ENTER RT
PASSWORD:
OK
PROJECT PASSWORD:

PROJECT NAME: REAL-TIME-GUIDE
@ABORT RECEIV
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G

*CLEAR-SEGMENT 244

RT-PROGRAMS ON SEGMENT:

  RECEIV
  SEND

DELETING THIS RT-PROGRAM(S)? Y
*EXIT
@FTN
NORD 10/100 FORTRAN COMPILER FTN-2090H
$COMPILE EX-7-S,,S
12 LINES COMPILED . OCTAL SIZE=     72
CPU-TIME USED IS 0.3 SEC.
$COMPILE EX-7-R,1,R
9 LINES COMPILED . OCTAL SIZE=     55
CPU-TIME USED IS 0.2 SEC.
$EXIT
@RT-LOADER

REAL-TIME LOADER,  SINTRAN III - VERSION G


*NREENTRANT-LOAD S,,
NEW SEGMENT NO: 244
*END-LOAD
NEGLECTING REFERENCES? N
*DECLARE-PROGRAM RECEIV
*END-LOAD
*NREENTRANT-LOAD R,,
NEW SEGMENT NO: 245
*END-LOAD
*EXIT
@
```

```
@RT SEND
@LOG
 16.22.42      30 DECEMBER 1981
 TIME USED IS   12  SECONDS OUT OF    5 MINUTES   2 SECONDS
---EXIT---

AASSDDFFGGHHQQWWEERRTTYY@

 16.23.10      30 DECEMBER 1981
 SINTRAN III - VS    VERSION G
ENTER RT
PASSWORD:
OK
PROJECT PASSWORD:

PROJECT NAME: REAL-TIME-GUIDE
@LIST-RT-DES RECEIV
I/O-WAIT  RING:0 PRIORITY:     20
LAST STARTED:     35 SECS
START ADDRESS:   2536, SEGMENTS:      0    244
P=     74
X=   3230
T=    200
A=   3230
D=     71
L=   2553
S=      0
B=   3410
READY
ACTUAL SEGM.:      0    244
RESERVED DATAFIELDS:
  33727

@ABORT RECEIV
```

APPENDIX B: RT programs in languages other than FTN

1 PASCAL

1.1 RT monitor calls

The Pascal run time library does not contain interface routines for RT monitor calls. However, ND Pascal may call Fortran routines, and routines available in FTNLIB may be called. The routine body of such routines will in the Pascal program consist of the sole word FORTRAN:

```
PROCEDURE RESRV(UNIT,INOUT,WAITFLAG: INTEGER); FORTRAN;
```

User written Fortran subroutines may be called in the same manner, but only non-reentrant rotines may be called from Pascal. That implies that the $REENTRANT-MODE must not have been given when the routine was compiled, and routines in the reentrant FTNRTLIBR cannot be loaded.

Where symbolic RT program names are submitted as argument, they should be of type PACKED ARRAY(.0..6.) OF CHAR. If the length is less than 7 characters, an apostrophe should terminate the name (as a character in the string).

Routines written in other languages (e.g. Planc) may be called, provided that the call interface is Fortran compatible (for Planc this implies that the routine modifier STANDARD must be present).

Code generated by the Pascal compiler is implicitly reentrant. No equivalent to the $REENTRANT-MODE command of the Fortran compiler is necessary or available.

1.2 File access

Files may be used in the same manner as in background. If parameters are missing in a CONNECT call, these are requested from the console (terminal 1). In order to prevent an error, device number 1 must have been explicitly reserved prior to this.

A device may be CONNECTed as a file, provided it has previously been reserved. Rather than specifying a file name, the device number is given as an integer constant or variable. If accessed through Pascal sequential I/O routines, the appropriate RESET or REWRITE calls must be executed prior to the first I/O call.

When the program terminates normally, file buffers are emptied and the corresponding files closed. Un-buffered files are not closed. This includes files opened with RX or WX access code.

The number of sequential files that are buffered can be set by the RT loader command

*DEFINE-SYMBOL NOBUF <no. of buffers> <segment number>

The default number of buffers is 4: the first four files concurrently opened for sequential access are buffered, the fifth and following files are not buffered. Each buffer is 256 words (1/4 page). The command

*DEFINE-SYMBOL NOBUF 0 <segment number>

may be used when loading programs not using sequential file access in order to save space.

1.3 Loading

Pascal requires special considerations when loading:

- The RT loader command

  *REFER-SYMBOL 5RTPM

  must be given prior to loading the library, in order to force routines suited for RT execution. This will cause RSIO (MON 143), callable as the integer function RUNMODE, to return 3, indicating RT execution.

  Also, errors occuring during execution are reported to the error device, because no terminal is implicitly reserved for the program. The error is written by ERMON, and has the form:

  ```
  11.19.22  ERROR  50  IN RTTEST AT 40322; USER ERROR
  SUBERROR:  30

  11.19.22  ERROR  51  IN RTTEST AT 40325; USER ERROR
  SUBERROR:  11
  ```

  The first message (error 50) indicates the Pascal error code as the suberror number; the second message (error 51) indicates the source line number. Pascal error codes are explained in "ND-100 Pascal Compiler User's Guide" (ND-60.124). An error will cause the program to terminate through RTEXT.

- As Pascal syntax does not allow for the specification of a program priority, it must be set explicitly. This may be done by either the Sintrant command @PRIOR or by the RT loader command *CHANGE-RT-DESCRIPTION.

- Statically allocated data and constants are loaded as common blocks. The starting address for these blocks should be defined through the command *PRESET-COMMON-ADDRESS. The address to be specified is dependent on the stack/heap space required: stack and heap is (by default) allocated in the area between the uppermost code address and the start of the common area. A trial load may be required in order to determine a reasonable address.

- The stack/heap area may (as in background) be located at an explicitly defined address, and have a user specified size. This is done by the commands

*DEFINE-SYMBOL STACK <lower address> <segment number>
*DEFINE-SYMBOL HEAP <upper address> <segment number>

These symbols must be defined before the PASCAL-LIB library is loaded. It is the responsibility of the user to ensure that the stack/heap area does not overlap any code or static data area. If the area is not between the code area (usually starting at address zero) and the common address, the *ALLOCATE-AREA command should be used to ensure that space is allocated on the segment. Otherwise, an error will occur when access to the area is attempted.

The PRESET-COMMON-ADDRESS and user defined stack/heap may be used to locate the data area on another segment than the code. The Pascal library is not completely reentrant, however. The Pascal library may be loaded to a separate segment used as a reentrant segment (MON REENT) by several programs. These programs must, however, have been loaded using the same address in the *PRESET-COMMON-ADDRESS command.

The command LOAD should be used to load code to the segment. The FTNLIB should always be loaded before the PASCAL-LIB, thus, it should be loaded explicitly. The REENTRANT-LOAD command should not be used.

The warning message: NO PRIORITY IN <prog name> may be ignored, but the priority of the program should be set before an execution is attempted. (Otherwise the program priority will be 1, which is usually lower than desired).

The RT program name will be equal to the first 7 characters of the name in the PROGRAM statement.

1.4 Example of loading

A complete job for loading the program RTX to segment 270 would be:

```
@RT-LOADER
CLEAR-SEGMENT 270
YES
NEW-SEGMENT 270,,,,,,,
REFER-SYMBOL 5RTPM
PRESET-COMMON-ADDRESS 270 20000
DEFINE-SYMBOL STACK 30000 270
DEFINE-SYMBOL HEAP  77777 270
ALLOCATE-AREA 270 50000 30000
LOAD PAS-OPEN,,,
LOAD FTNLIBR,,
LOAD PAS-LIB,,
END-LOAD
CHANGE-RT-DESCRIPTION RTX 61,,,,,,
END-LOAD
EXIT
@RT RTX
```

The code size of the program including libraries does not exceed 20000B words, the static data area does not exceed 10000B words, and 50000B words are allocated for stack and heap for a total of 32 pages.

2 FORTRAN-100

The RT loader has certain features oriented towards the FTN compiler, particularly regarding reentrant Fortran programs. These are not applicable to the new ANSI-77 standard compiler FORTRAN-100.

2.1 Unit numbers for input/output

FORTRAN-100 will translate the Fortran file number to a Sintran file number even in reentrant programs. Thus, the file number may be a constant in the range 1:127, as well as an integer variable.

For most calls, if a file with the specified number is not opened when the I/O operation is performed, the number is assumed to be a Sintran device number. The device must be reserved when the program is executed; the reservation may be performed by commands or by another program (PRSRV call).

The Sintran device/file number corresponding to a Fortran unit number can be obtained through the integer function LDN:

```
OPEN(6,FILE='LINE-PRINTER',ACCESS='SEQUENTIAL')
LOGDEV = LDN(6)
```

2.2 Input/output buffers

When accessing disk files, the I/O system will use buffers rather than byte-by-byte access if possible. These buffers are allocated in the stack area of the program.

In **non-reentrant** mode, the buffers are allocated in the space following the last code loaded. The number of 1K buffers allocated is determined by the space between the uppermost load address and the lowermost common address. Buffers are allocated when the first access to the file is done, and deallocated when the file is closed. If more files are in use concurrently than there are buffers, the files last taken in use will be accessed byte-by-byte (unbuffered).

If no common data is loaded, no space above the uppermost load address will be allocated to the segment. When a disk file is accessed, this will cause a PAGE FAULT FOR NON-DEMAND error. In order to prevent this, the area must be explicitly allocated and the start of the (empty) common area defined. Assuming segment number 213 and maximum 3 files concurrently opened (6000B locations for buffers) and a total code size of 31200B words, the commands required are:

```
*PRESET-COMMON-ADDRESS 213 40000
* <load code and library>
*ALLOCATE-AREA 213 6000B,,
```

In **reentrant mode**, each program has its own buffers in the stack area, and the total size of buffers required must be considered when allocating the stack. The maximum number of buffers is determined by

an explicit call to the routine:

```
CALL CREBUF(n)
```

"n" is the number of buffers allocated. Only the first call to this routine has any effect. If "n" is negative or zero, no buffers are allocated, and all files are accessed in a byte-by-byte fashion.

2.3 Stack allocation in reentrant-mode

No automatic allocation of the stack is possible with the FORTRAN-100 compiler. The stack usually follows the Fortran library, and is of size 5STLEN. This symbol must be defined by the user before *END-LOAD. As with file buffers, the segment area is not allocated unless COMMON areas have been loaded at a higher address than the uppermost stack address (load address after library is loaded, plus stack length). The *ALLOCATE-AREA command may be required in order to ensure that no illegal page faults occur.

The size of the stack depends on the program, but the compiler will report for each compilation the stack size required. The demands of all routines that will be active concurrently may be added together to estimate the minimum total. If a routine is called recursively, the stack requirement for that routine is the value reported by the compiler times the maximum recursion level.

The stack may be allocated at a non-default address by defining the symbol 5STBEG before the Fortran library is loaded. The definition of 5STBEG and 5STLEN does not imply any allocation of segment space, and the user is responsible for setting load addresses and allocating area to ensure 1) that no code is loaded in the stack area, 2) that the area is actually allocated.

2.4 Conflicts with other libraries

If default stack allocation is used (stack immediately above the Fortran library), no code may be loaded after the Fortran library has been loaded. That would cause the code to be loaded within the stack area. Thus, the library must be the last file loaded.

Some other libraries have similar requirements, for example the Planc library if the Planc symbol FREE_P is used. Thus, collisions may occur. These can be avoided by two different methods:

- the Fortran stack may be explicitly allocated before the library is loaded
- after the Fortran library but before other libraries are loaded, the current load address may be set by the *SET-LOAD-ADDRESS command to an address above the Fortran stack.

2.5 The Fortran library

The same library is used for reentrant and non-reentrant routines. However, the 2-bank facility available in background is not available, and the use of the alternative page table in the 2-bank library may cause havoc if used with RT programs. Thus, only the 1-bank version of the library should be used.

The old FTNLIBR is not compatible with the new compiler.

2.6 Loading

FORTRAN-100 code is loaded by the *LOAD command. The *REENTRANT-LOAD command may not be used, as it will automatically load some routines from the old FTNRTLIBR, which are incompatible with FORTRAN-100.

As mentioned above, a program using I/O facilities will require buffers and the common address must be preset. If no common is loaded explicit allocation of segment area is required.

3 BASIC

3.1 Program compilation

All programs must be compiled to a :BRF file in order to be loaded by the RT loader; incremental compilation is not possible for RT programs.

An EOF statement should be the last one in the source program (including files containing subroutines only).

3.2 Priority notation

A decimal priority may follow the PROGRAM statement; the value should be followed by a per cent sign to force it to integer type:

```
PROGRAM PRRT, 30%
```

If no priority is given, the RT loader will assign the program a priority of 1; in most cases the programmer will then set the priority through the @PRIOR command.

3.3 RT monitor calls and Fortran routines

The Basic library (BASLIBR) does not contain interface routines to RT monitor calls, but they may be loaded from the non-reentrant Fortran library (FTNLIBR).

User written Fortran routines compiled with the FTN compiler may be called from Basic, provided they are compiled in non-reentrant mode. Reentrant Fortran routines may not be called, as the stack mechanisms in Basic and Fortran are incompatible.

BASLIBR and FTNLIBR has several common entry points, while the corresponding code is different. In general, where FTNLIBR must be scanned due to an explicit call in the Basic program, BASLIBR should be loaded prior to FTNLIBR.

There is no "reentrant" version of the Basic library. On special request, a library split on two files may be obtained from Norsk Data: one contains the non-reentrant routines that must be loaded with each program, another contains the reeetrant routines that may be loaded to a separate segment and linked to by all Basic RT programs. This will save approximately 8 pages per Basic RT program.

3.4 File access

If a file must be opened for both read and write, Basic requires explicit calls for read and write access, through two separate OPEN statements. Two different connect device identifiers are then used for the same file. This will fail, as the file reservation in the second OPEN will overwrite the first. Under such circumstances, the OPEN statment must be accompanied by explicit RESRV calls, which require that the logical unit number is known.

```
 20 PROGRAM TEST,20,%
 30 OPEN #1: FOR INPUT  "T4010"      'T4010 = TEKTRONIX, LDN=7
 40 OPEN #2: FOR OUTPUT "T4010"
 50 CALL RESRV(7%,0%,1%)             'EXPLICIT RESERVATION NEEDED
 60 PRINT #2: " TEST2"!
 70 INPUT #1: I
 80 PRINT #2: !" I=",I
 90 CLOSE #1:
100 CLOSE #2:
110 END
120 EOF
```

All files opened by a Basic program should be explicitly closed after use. Files are not closed by the END statement, and if the files are closed by an @RTCLOSE command, buffers for sequential output files are not emptied properly.

3.5 PRINT and INPUT without connect identifier specified

If no connect identifier is included in the the PRINT and INPUT statements, in background the I/O is performed to the user terminal (logical device number 1). In RT programs, this will instead refer to the system console.

It is therefore recommended to use only PRINT #n: and INPUT #n:, with connect device identifiers specified. If I/O to the console is required, device number 1 should be explicitly reserved.

3.6 Peripheral devices

Peripheral devices may be accessed directly by their logical device numbers used as connect device identifiers. The deviced must then be reserved and released explicitly.

3.7 Loading

The :BRF files may be loaded by either of the commands *LOAD and *NREENTRANT-LOAD, but BASLIBR must be explicitly loaded before the loading session is terminated.

Basic uses an area (stack) for dynamic run time allocation of data space. This area is in background located between the code and the common blocks. If the program uses common blocks the user need not be concerned about the dynamically allocated space. Even if no common blocks are used, the start address of the common area must be specified by the command *PRESET-COMMON-ADDRESS, and the area from the

uppermost code address to the common address must be explicitly allocated through *ALLOCATE-AREA.

The size of the stack is estimated approximately as for a Fortran program. BASLIBR requires roughly 1000B locations, and the total size of variables, matrices etc. must be added.

Basic programs using common blocks are handled like Fortran programs using COMMON: in order to allocate the common blocks adjecently rather than scattered among the code, the *PRESET-COMMON-ADDRESS is used before loading. However, this command should, due to requirement of the Basic stack, always be used even if the program uses no common blocks.

4 PLANC

Planc will not create special probles when loaded with the RT loader.

Programs should be compiled with the DEBUG option off.

The SEPARATE-CODE-DATA option is legal but the RT loader is incapable of loading code and data to separate segments. Full separation of code and data is most easily obtained by declaring all data in a separately compiled module that may be loaded to any segment independent of the code. Under no circumstances should the 2-bank run time library be used; the programmer is responsible for all manipulation of the alternative page table.

Planc code is loaded by the *LOAD command. As the Planc stack usually is a declared array, no problems occur with unallocated stack space.

If the FREE_P pointer is used, the required heap space must be allocated during load time.

APPENDIX C: Interface to assembler routines

This appendix gives sufficient information for a programmer experienced in MAC assembler programming to write MAC routines for interface to monitor calls not available in the FTN, FORTRAN-100 and Pascal libraries. All other languages suitable for RT programming are able to interface to subroutines using a FORTRAN-100 call sequence.

1 Pascal

The Pascal programmer has two options: he may either declare his routines as EXTERNAL and use the Pascal call sequence, or he may declare them as FORTRAN and use the call sequence described below for FORTRAN.

This section will describe the Pascal call sequence.

1.1 Register contents

Upon entry to an EXTERNAL routine the registers contain

X - static link
A - address of new stack frame relative to B
B - previous stack frame (dynamic link)
L - return address

The T and D registers are not assigned specific functions.

1.2 Stack frame

Adding the B and A register gives the absolute address of the new stack frame. The stack grows from low towards higher addresses. The stack head consists of three locations plus the function value:

A + B : Static link
A + B + 1 : Dynamic link
A + B + 2 : Return address
A + B + 3 : function value (if any)

The function value may occupy 0 words (procedure), 1 word (integer, char, subrange, pointer), 2 words (longint, real if 32 bit hardware) or 3 words (real if 48 bit hardware).

1.3 Parameters

The procedure/function parameters follow the function value. In case of a VAR parameter, the address is transferred, in case of a value parameter the value itself is found on the stack. A value parameter may occupy from 1 to 8 words on the stack. If the value is occupies more than 8 words, the address of a copy of the value is found on the stack.

For example, if a Pascal fuction is declared as

```
FUNCTION FN(VAR I: INTEGER; F1: REAL; VAR F2: REAL): INTEGER;
EXTERN;
```

the stack has the following layout

```
A + B -->   static link
      + 1 : dynamic link
      + 2 : return address
      + 3 : integer function value
      + 4 : address of I
      + 5 : F1 real value
      + 6 :     "
      + 7 :     "
      + 8 : address of F2
```

The Pascal heap grows from high addresses towards low. The calling program will guarantee that at least 200B locations of stack space are available upon entry to an external routine, thus, addresses up to A + B + 177B may be used freely by the MAC routine.

1.4 Routine exit

On exit, a function value must be loaded to the A, AD or TAD registers (the function value location on the stack need not be filled in by the external routine, but is used internally within Pascal).

The B register must contain the same value as it had on entry. Return is to the address found in the L register on entry (thus, the MAC instruction EXIT may be used if the L register has not been modified.

The contents of the X register and of T, A and D if not used for function value, is arbitrary.

As an example, a MAC routine to allow a Pascal program to set the break mode of an arbitrary terminal (the Pascal library routine will set break mode for the current background terminal only):

```
PROCEDURE XBRKM(DEVNO: INTEGER; MODE: INTEGER);
EXTERN;
```

```
)9BEG
)9ENT XBRKM

        BRKM=4

XBRKM,  RADD SB DA
        COPY SA DX          % Start of local data area
        LDT 3 ,X            % Device no to T register
        LDA 4 ,X            % Break mode to A register
        MON BRKM            % Always direct return
        EXIT                % Return to Pascal prog
)9END
)9EOF
)LINE
```

2 FORTRAN-100

2.1 Register contents

Register contents on entry to a subroutine or function:

T - atual number of parameters
A - parameter list address
D - address of result string descriptor if CHARACTER function, otherwise unused
X - unused
B - current stack element
L - return address

The actual number of arguments in the T register allows a MAC routine to substitute default values or take other action depending on the number of parameters supplied.

2.2 Parameters

Essentially, parameter transfer is as for RT monitor calls: the A register points to a argument list which contains the addresses of the actual argument values.

A multiple word value, e.g. a real value, is addressed by its lower word in memory (the exponent part).

An RT description address is treated as a variable; in the argument list is found the address of a location containing the RT description address.

An array is transferred like a single variable; the argument list contains the address of the first variable in the array. If the lower index is not 1, the address of the element with the lowest index is transferred.

A CHARACTER parameter is transferred by a two word descriptor whose address is found in the parameter list. The first word of the descriptor contains the address of the first character. The second word consists of three subfields:

- bit 17B : = 0 string starts in left byte
  = 1 string starts in right byte
- bits 16B:13B: Used by CE option, should normally be 0
- bits 12B:0 : Length of string in bytes

For CHARACTER arrays the parameter list contains the address of a descriptor of the first string in the array, and the length indicated is the length of a single element of the array.

2.3 Stack element

The stack element starts at B-200B, and has the following layout:

```
B-200B: LINK    Return address
B-177B: PREVB   Previous B register, reloaded on exit
B-176B: FREES   First free stack address above this element
B-175B: EOS     First address beyond stack
B-174B: SYS     Used by Fortran runtime system
B-173B: ERRCODE value
```

In REENTRANT-MODE the parameter list is allocated from B-172B and upwards; the length of the list is found in the T register. Stack elements are dynamically allocated on the stack.

The stack layout is the same in nonreentrant mode, but is allocated in a fixed location; FREES will be unmodified. The parameter list is allocated in the local data area of the calling routine.

2.4 Function value

A LOGICAL, INTEGER, INTEGER*4 or REAL function value is returned in the registers (TAD, AD or A).

A REAL*8, COMPLEX or COMPLEX*16 value is returned in a location pointed to by the A register.

A CHARACTER function value is stored in the string whose descriptor address is found in the D register on entry. The function must not store a longer string than the length part in the descriptor allows, and must not modify the descriptor.

2.5 Routine exit

If the B register is modified by the routine, it must be restored before return.

Return is always to LINK (which is also found in the L register). If the parameter list contains alternate returns, the alternate return value is stored in the ERRCODE position in the caller's stack element (this value may be used in a computed goto after return to the caller). The alternative return addresses are omitted from the parameter list. Ordinary return is indicated by the value 0.

3 FTN

Most of the description of the call sequence for FORTRAN-100 applies to the FTN compiler. CHARACTER descriptors and stack layout are different.

3.1 CHARACTER descriptor

The descriptor is a two word entry; the first word is the address of the string, the second consists of

- bit 17B : = 0 string starts in left byte
  = 1 string starts in right byte
- bits 16B : Must be 1
- bits 15B:0 : Length of string in bytes

3.2 Routine entry

In REENTRANT-MODE, the stack demand (stack header, 15B locations, plus what is needed for arguments and local variables) follows the entry point. The entry point should therefore be a JMP*+2 instruction. The parameter list is located at B-163, following the stack header.

The stack header has the following layout:

3.3 Function value

LOGICAL, INTEGER, INTEGER*4 and REAL function values are returned in the TAD, AD or A register.

A CHARACTER function value descriptor is found in the AD register.

A COMPLEX function value is returned in the TAD (48 bits hardware) or AD (32 bits hardware) register for the real part, B-172B, B-171B and B-170B (48 bits hardware only) for the imaginary part.

3.4 Routine exit

To free the stack space, return should be performed by a jump to the library routine 8LEAV, declared as an external symbol ( )9EXT 8LEAV). The B register must have the value it had on entry.

4 Planc

4.1 Assembler routines for Planc programs

Mixing of MAC and Planc is in general discouraged. Where access to monitor calls or instructions not accessible in Planc is needed, inline assembly code is usually better suited and easier in use.

Composite arguments - arrays and records - are always transferred by address, as for Fortran. However, non-WRITE simple parameters are transferred by value; the actual parameter value rather than its address is found in the parameter list. A WRITE simple parameter is copied to the stack on entry, and copied back to the caller on exit. The MAC routine will therefore be dependent on the IMPORT declaration in the Planc program.

A routine declared as ROUTINE STANDARD will transfer all parameters by reference; the MAC routine will be Fortran as well as Planc compatible. Return is to the first location following the call. Neither STANDARD nor REFERENCE allows in-values. Planc stack layout is the same as for FORTRAN-100.

In-values are are transferred like a function value: a simple variable is found in the TAD, AD or A registers; a compound value is transferred by its address in the A register. (The argument list is always found at B-172B if no routine modifier is present; the A register is therefore available.)

4.2 Planc routines for Fortran or Pascal

A ROUTINE STANDARD may be called from any language that may call Fortran compatible routines.

A Planc routine called from a non-Planc program should always initialize its own stack by INISTACK. Although the stack header layout is the same as for FORTRAN-100, there are slight differences in the stack handling. Planc routines may be used from REENTRANT-MODE as well as non-reentrant Fortran.

A Planc ROUTINE STANDARD used by a Pascal program should be decalred in the Pascal program like a Fortran subroutine or function:

```
PROCEDURE NAME; FORTRAN;
```

An example of a Planc routine used by a Fortran program is shown in chapter 20.

APPENDIX D: Loading a SINTRAN RT system

In order to load user programs in :BRF format into an Sintran RT system, the code can be inserted into the :BPUN file containing the Sintran RT operating system on a Sintran III/VS/VSE system.

RT Sintran is delivered on a floppy containing two identical files, one of type :IMAG, one of type :BPUN. The :IMAG file will usually be kept unmodified as a backup, while user RT programs are loaded into the :BPUN file. As the :BPUN file is the only :BPUN file on the floppy, it will be loaded automatically when the floppy monitor is started after a "1560&" microprogram command, or the LOAD button pressed (assuming that the ALD switch register is set to 1560B).

The loading of code into the :BPUN file is done through the command

*IMAGE-LOAD <image file> <output file>
(<RT description size>) (<bootstrap addr.>)

Command parameters:

<image file> - file containg the original RT Sintran. Default file type is :BPUN.

<output file> - file that will contain the new copy of the operating system with the user programs added

<RT descr. size> - size of RT description; should be 32 for the RTP version, 24 for the RT version. Default is 24.

<bootstrap addr.> - address of the BPUN bootstrap. Default is current load address at END-LOAD.

The <image file> must contain an RT Sintran image. Observe that the default type of this file is :BPUN, while the file ordinarily used as backup copy is delivered with :IMAG as type.

It is possible to add more programs to an already modified Sintran file, by giving this as <image file>. The <image file> and <output file> may be the same file.

After this command is issued, subsequent commands will refer to the RT Sintran image, rather than to the system used for loading. Code will be loaded to the image rather than to a segment. Symbols definitions and references apply to the image, not to the RTFIL, and RT description addresses are found in the tables in the image.

A number of commands are not relevant to RT Sintran. E.g., there is no segment file, programs are located contiguously in memory, and will always be resident. All commands relating to segments and files are illegal. These will cause the error message THIS COMMAND NOT ALLOWED NOW. The list of illegal commands is found in appendix D of SINTRAN III Reference Manual ND-60.128.

After all programs have been loaded, the current load address must be patched into location 30, the address of the first free RT description into location 32. This is done by the *CHANGE-LOCATION command:

```
@LIST-FREE-RT-DESCR,,,

  31275  31327  31361  31413  31445  31477  31531  31563
  31615  31647  31701  31733  31765  32017  32051
@CHANGE-LOCATION
32/ 30671  31275
  26711  .
@WRITE-LOAD-ADDR
L.ADR:        0   U.ADR: 101673   C.LADR: 101674
@CHANGE-LOCATION
30/ 101645  101674
  100211  .
@
```

Symbolic names of RT programs are not stored, but may be defined after the RT Sintran is started by the command @DEFINE-PROGRAM, corresponding to the *DEFINE-PROGRAM in the RT loader.

********** SEND US YOUR COMMENTS!!! **********

Are you frustrated because of unclear information in this manual? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card — and an answer to your comments.

Please let us know if you

* find errors
* cannot understand information
* cannot find information
* find needless information

Do you think we could improve the manual by rearranging the contents? You could also tell us if you like the manual!

********** HELP YOURSELF BY HELPING US!! **********

Manual name: SINTRAN III Real Time Guide

Manual number: ND-60.133.02 Rev. A

What problems do you have? (use extra pages if needed) _____

_____

_____

_____

_____

Do you have suggestions for improving this manual ? _____

_____

_____

_____

_____

_____

Your name: _____ Date: _____

Company: _____ Position: _____

Address: _____

_____

What are you using this manual for ? _____

_____

NOTE!
This form is primarily for documentation errors. Software and system errors should be reported on Customer System Reports.

Send to:
Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
0621 Oslo 6, Norway

Norsk Data's answer will be found on reverse side

Answer from Norsk Data _____

Answered by _____ Date _____

Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
0621 Oslo6, Norway

Systems that put people first