Report No. SAT88-0002

SPACE STATION

OPERATING SYSTEM STUDY

SUMMARY REPORT

PREPARED FOR NASA
UNDER CONTRACT
NAS8-36462

PREPARED BY:
ALBERT E. HORN
MORRIS C. HARWELL
SMITH ADVANCED TECHNOLOGY, INC.
HUNTSVILLE, ALABAMA

February 1988

SMITH ADVANCED TECHNOLOGY, INC.

SPACE STATION OPERATING SYSTEM STUDY

SUMMARY REPORT

LIST OF TABLES

LIST OF FIGURES

SPACE STATION OPERATING SYSTEM STUDY

SUMMARY REPORT - FEBRUARY 1988

I. OVERVIEW AND SUMMARY

A. Purpose and Methodology

The current phase of this study has been based on the analysis, evaluation, and comparison of the operating systems implemented on the computer systems and workstations in the software development laboratory. Primary emphasis of the study has been the DEC MicroVMS operating system as implemented on the MicroVAX II computer, with comparative analysis of the SUN UNIX system on the SUN 3/260 workstation computer, and to a limited extent, the IBM PC/AT microcomputer running PC-DOS. Some benchmark development and testing was also done for the Motorola MC68010 (VM03 system) before that system was removed from the lab. These systems were studied with the objective of determining their capability to support space station software development requirements, specifically for multi-tasking and real-time applications. The methodology utilized consisted of development, execution, and analysis of benchmark programs and test software, and the experimentation and analysis of specific features of the systems or compilers in the study.

B. Systems Tested

Since the Ada programming language is the language selected for Space Station software use, the primary programming language used for test software development on these systems was Ada. Other languages were used as a comparison where compilers were available, such as FORTRAN, "C", and Pascal. Several times during the course of the study, newer releases of compilers were received and installed on the systems. In such cases, all benchmarks and test programs were rerun to obtain new data, such that all data presented in this report is a result of the most recent releases. A table of the computer systems that were included in the study, their operating systems, languages, and software versions is presented in Table I-1.

COMPUTER	OPERATING SYSTEM	LANGUAGES
DEC MicroVAX II	MicroVMS 4.5	VAX Ada V1.3
		VAX FORTRAN V4.5
		VAX C V2.2
		VAX/VMS Macro V04
SUN 3/260	SUNOS 3.2	SUN FORTRAN 77
	(UNIX 4.2 BSD derived)	SUN C
		SUN Pascal
		Alsys Ada 3.0
		SUN Assembly
IBM PC/AT	PC-DOS 3.2	Alsys Ada 3.2
Motorola MC68010	UNIX System V/68 2.1	TeleSoft Ada 2.1
(VM03)		Sys V/68 FORTRAN 77
		Sys V/68 C
		Sys V/68 Assembly
	VERSAdos 4.3	VERSAdos Pascal
		M68000 Assembly

Table I-1. Systems Tested

C. Structure of Report

In the following sections of this report, the study is presented in its major categories. First, benchmark programs are detailed. These programs were developed to quantify the performance of selected functional areas of the operating systems/languages and produce a measured execution time as a result. Second, test programs are documented. These programs were used to test certain features and capabilities of the operating system or language, and as such did not produce a measured result. They did produce an observable behavior from which conclusions could be made or the operating systems could be compared. Next, specific operating system/compiler analysis results are documented. For these cases, special tests to analyze a particular problem area or to test options unique to a particular Ada compiler or operating system were made. Finally, the source code listings for the software developed for the study are presented. These listings should prove valuable not only to allow comparisons to be made for other operating systems/compilers, but to serve as an example of implementation techniques for some of the features available with the systems tested, such as calling operating system services, or linking "foreign" subprograms to an Ada main program.

D. Benchmark Summary

A combined summary of benchmark timing results of all the benchmarks for all the systems tested, for each programming language used, is presented in Table I-2.

SPACE STATION OPERATING SYSTEM STUDY BENCHMARK SUMMARY
TIMING RESULTS (ALL TIMES IN SECONDS)

BENCHMARK		**SYSTEM**		
	MicroVAX II	SUN 3/260	IBM PC/AT	Motorola 68010
Sieve Prime Number Execution				
FORTRAN	1.1 (4 byte integers) 1.7 (2 byte integers)	.50		13.8
C	1.3	.67		12.3 5.7 (optimized)
PASCAL		.62		6.8 (VERSAdos)
ADA	1.5 1.2 (w/suppress_all)	.89 .66 (check= stack)	5.1 3.7 (check= stack)	205.8
ASSEMBLY	1.2 (2 byte integers)	.39		4.5 (UNIX) 2.6 (VERSAdos)
Sieve Prime Number Compile and Link				
FORTRAN	2.6 (compile) 7.5 (link)	6.0		
C	6.1 (compile) 35.4 (link)	3.2		50.8
PASCAL		5.8		
ADA	14.5 (compile) 16.2 (link)	25.0	56.7 (compile) 44.3 (link)	735.2
ASSEMBLY	15.2	2.0		

Table I-2 Benchmark Summary

	MicroVAX II	SUN 3/260	IBM PC/AT	MOTOROLA 68010
Floating Point Subroutine (no library)				
FORTRAN				
single	3.3	6.2		399.3
double	4.2 (d-float)	15.3		
	4.2 (g-float)			
quad	462.7			
C				
single	6.0	18.0		205.7
double	5.0 (d-float)	15.2		
	5.0 (g-float)			
quad	not supported			
Ada				
single	1.8	6.8 (float= M68881)	27.7	300.6
double	2.7 (d-float)	10.0		
	2.7 (g-float)	8.0 (float= M68881)	45.5	
quad	553.5			
Floating Point Library Sine Function				
FORTRAN				
single	1.8	7.4		144.9
double	3.3 (d-float)	11.6		
	58.9 (g-float)			
quad	266.7			
C				
single	3.6	12.2		142.6
double	3.5 (d-float)	11.9		
	59.2 (g-float)			
quad	not supported			
Ada				
single	1.7	4.4		
		4.0 (float= M68881)	8.7	no math lib
double	3.3 (d-float)	9.6	9.9	no math lib
	58.7 (g-float)	7.0 (float= M68881)		
quad	276.2			

Table I-2 (continued)

	MicroVAX II	SUN 3/260	IBM PC/AT
MATRIX			
FORTRAN			
single	9.7	11.6	
	6.5 ("C" algorithm)		
double	11.0 (d-float)	19.7	
	11.1 (g-float)		
quad	518.8		
C *			
single	8.6	20.3	
double	6.9 (d-float)	16.9	
quad	not supported		
Ada			
single	10.0	13.9	71.2
	7.4 (w/suppress_all)	13.3 (float= M68881)	61.0 (checks=stack)
double1	11.1	15.8	76.8
	8.6 (w/suppress_all)	16.3 (float= M68881)	67.0 (checks=stack)
quad	403.6		
	398.6 (w/suppress_all)		

* C does not support variable dimensions, a simulated algorithm was used.

Table I-2 (continued)

	MicroVAX II	SUN 3/260	IBM PC/AT	MOTOROLA 68010
Disk Write Timing				
FORTRAN	9.8 (pre-allocated file blocks)	4.0 (buffered)		57.0
C	10.0 (pre-allocated)	2.0 (buffered)		39.0
PASCAL		2.0 (buffered)		19.0 (VERSAdos)
Ada	9.7 (pre-allocated & contiguous)	1.9	13.4	53.4
VAX/RMS (from FORTRAN)	8.3 (buffercount=22)			
VAX/RMS (from FORTRAN)	12.2 (block I/O, <u>not</u> buffered)			
Ada Tasking				
Rendezvous	1.7	0.6	1.0	129.0
Send/Receive	6.6	3.2 2.6 (checks= stack)	19.1 16.9 (checks= stack)	775.7
Send/Receive (shared disk)	50.0 33.0 (pre-allocated & contiguous)	7.2 (buffered)	58.3	

Table I-2 (continued)

Send/Receive 2-Task Data Transfer Benchmark

MicroVAX II Using VMS System Services

Installed Shareable Common for data transfer:	
with event flags	8.1
with hiber/wake	8.1
with suspend/resume	8.4
with shared common flags/.01 sec wait*	9.7
with lock/unlock/.01 sec wait*	12.1
Mapped Global Section Common for data transfer	
with hiber/wake	7.5
Mailbox for data transfer and sync.	13.3
Shared Disk File for data transfer	
with shared common variables	20.7

*.01 second is smallest wait time available

SUN 3/260 Using UNIX System calls

Shared Memory for data transfer:	
with signals/.1 sec delay*	60.1
with semaphores/.1 sec delay*	60.1
with shared memory flags/.01 sec delay	6.2
with shared memory flags/no delay	50.5

*shorter delays caused loss of synchronization

Table I-2 (continued)

Process Creation Benchmark

MicroVAX II Using VMS System Services,	time in seconds per each process creation
Spawn Subprocess:	
with defaults	.88
with installed process image	.82
with installed process image, no optional data structures	.80
Create Detached Process:	
with hiber/wake synch., .07 sec wait	.32
with event flag synch., .07 sec wait	.32
with termination mailbox, synch. read to mailbox	.31
with termination mailbox, synch. read to mailbox, installed image	.28

Synchronization Benchmark

MicroVAX II	
with hiber/wake services	1.2 seconds

Table I-2 (concluded)

E. Source Code Conventions

All source code files generated as a result of this study follow standard naming conventions with regard to file extension (file type) for each system and each language used. To form the source code file name in order to examine the source code, merely append the program name given in the benchmark program and test software descriptions with the appropriate file type. For example, the disk timing benchmark, named "timtes" is appended with the file type, ".ada" to form "timtes.ada", which is the file name of the source code for this benchmark. A table of all file extensions or file types for each of the systems studied, for each language used, is presented in Table I-3, using the benchmark program named "zprime" as an example.

COMPUTER	PROGRAM NAME
Motorola	
FORTRAN	zprime.f
Pascal	zprime.sa
"C"	zprime.c
Ada	zprime.text
Assembler(UNIX)	zprime1.s
Assembler (VERSAdos)	zprime1.sa
MicroVAX II	
Ada	zprime.ada
FORTRAN	zprime.for
Assembler	zprime.mar
"C"	zprime.c
IBM PC AT	
Ada	zprime.ada
SUN 3/260	
FORTRAN (F77)	zprime.f
"C"	zprime.c
Pascal	zprime.p
Assembler	zprime.s
Ada	zprime.ada

Table I-3 Source Code Conventions

F. Conclusions

1. Ada

The performance of the Ada compilers as used with the operating systems and computers included in the study have proven to be on a level with the other high-level languages to which they were compared, with the exception of the TeleSoft Ada compilers used on the Motorola System V/68 UNIX operating system. The TeleSoft compilers (1.5 and 2.1) were preliminary versions whose performance was so poor that they were useless as software development tools, and should now be considered obsolete. Among the remaining Ada compilers tested, the DEC VAX/VMS compiler is clearly superior. VAX/VMS Ada is very well integrated into the VMS operating system environment, having complete access to the rich set of VMS system services and system libraries. No "bugs" or anomalies were found with the VAX/VMS Ada compiler. In addition, the documentation for VAX/VMS Ada is excellent, consisting of several volumes covering the complete Ada software development system. Even DEC's version of the Ada Language Reference Manual has been enhanced with VAX/VMS-specific information.

The Alsys Ada compiler was also very good, although not quite as mature as the DEC VAX/VMS product. Some errors were found with the version of the compiler that Alsys delivered for the SUN workstation. Alsys, Inc. claims that these problems are supposed to be corrected in later releases. The problem areas are discussed in Section IV of this report. It should also be noted that version 3.2 of the Alsys Ada compiler as delivered for the PC/AT finally included a math library, but this library is a user supplied library and is not supported by Alsys, Inc.!

During the course of the study, the Ada language itself has proven to be a capable language for real-time and multi-tasking applications, frequently obviating the need or supplanting the requirement for direct calls to operating system services. This is especially true for UNIX or UNIX derived time-sharing operating systems, or for operating systems with no direct capability for real-time or multi-tasking functions, such as PC-DOS. In fact, the send/receive benchmark for the MicroVAX showed that in some cases, execution time for Ada tasking and rendezvous is faster than equivalent methods using operating system services. There were, however, some problem areas uncovered with the use of Ada for these applications. For example, the capability for one task to suspend and resume the execution of another is lacking in Ada. This was demonstrated with the multi-task time-shared execution test program (task_exec) where either a complex time-checking subprogram "work-around" had to be used, or a time-slice option relied on, if available. For VAX/VMS, operating system services exist to perform this function as an alternative, while on a UNIX system, they do not. Thus, where a real-time application requires scheduling of tasks, Ada's current inability to provide task control of other tasks' execution would be a limitation to be considered.

2. Operating Systems

Of the operating systems considered in this study, the most applicable by far for real-time, multi-tasking applications is MicroVMS for the DEC MicroVAX II. VAX/VMS (or MicroVMS, which is actually the same operating systems as VMS for the larger VAX computer systems, only tailored specifically for the MicroVAX) is DEC's standard proprietary operating system for the VAX series of computers.

An important measure by which to judge an operating system's suitability for real-time or multi-tasking use is its programming interface, as opposed to its user interface, (which is a greater factor in determining how "friendly" it is for the human user to interact with the system). The programming interface for VAX/VMS consists of an extensive repertoire of system services, standard libraries, and optional routines all callable from any of the standard VAX/VMS programming languages. The system services offer a variety of means for a VMS process to communicate with another process, control another processes' execution, control its own execution, exercise direct control over I/O devices, control system resources, and utilize system utilities.

The SUN workstation's UNIX operating system also has a considerable number of operating system services (or "system calls" as the UNIX documentation refers to them), and an extensive library. The system calls do not, however, have as extensive a process communication and control capability as VAX/VMS. For example, there is no direct SUN UNIX call for a process to suspend or resume the execution of another process. SUN UNIX system calls and libraries are a combination of those from AT&T UNIX System V and Berkeley UNIX 4.2 BSD and as such do not incorporate any real-time extensions as offered by vendors of "real-time UNIX" systems. A more complete description of the UNIX operating system capabilities can be found in the study report for Phase I of this study, titled "Space Station Operating System Study Phase I Report".

II. BENCHMARK SOFTWARE

The following subsections describe the benchmark programs.

A. Prime Number Benchmark

Typically, the first benchmark program to be applied to any system under test was the "Sieve of Eratosthenes" prime number calculation program derived from Byte magazine, September, 1981. This program is based on using only repetitive addition for its algorithm, no multiplication or division, and tests a system's high-level language capability as well as the operating system and processor performance. This program (name: zprime for all except Motorola UNIX assembly: zprime1.s, and Motorola VERSAdos assembly: zprime1.sa) was coded and executed for the widest variety of languages of any of the benchmarks. It was also used to measure compilation and linking (or binding) speed for each case tested. Both the execution and compile/link timing data is presented in the summary section of this document. A highly commented FORTRAN version of this benchmark is presented in the source listing of Figure II-1 to explain the algorithms used in the program. Although this version is correct, it was not used to obtain timing data and is included here only for documentation.

B. Floating Point Benchmarks

The floating point benchmarks were derived from floating point test programs published in DEC Professional magazine, September 1986 and December 1986, and serve the purpose of testing the floating-point capabilities of the system. These benchmarks consist of two programs. One, (fpatestsub) to repeatedly perform floating point

(zprime_pretty.for)

```
!
      PROGRAM ZPRIME

c   zprime.for   Eratosthenes Sieve Prime Number Program in FORTRAN.
c                Compute all primes from 3 to 16K


      logical flags( 8191 )     ! Array representing odd numbers starting
                                !      with the number 3 and ending with
                                !      the number 16383.
      integer i                 ! Index for flags array.
      integer iprime            ! Used to store the actual value of a prime.
      integer k                 ! Index in flags array for odd multiples
                                !      of a prime.
      integer icnt              ! Total number of primes.
      integer iter              ! Number of iterations of the program.



      write( 6,900 )
 900  format(' 10 iterations')

c                              Repeat the calculations 10 times so we can
c                              measure the time.
      do 92 iter = 1,10

c                              Set count of number of primes to zero.
         icnt = 0

c                              Set 8191-element array of flags (representing
c                              all odd numbers starting with the number 3 and
c                              going thru 16383) to all true (true meaning the
c                              number is a prime).  When finished with sieve,
c                              all non-prime odd numbers will have their flags
c                              set to false.  We must start with the number 3
c                              because we know that only after 2, all primes
c                              are odd numbers!
         do 10 i = 1,8191
 10          flags( i ) = .true.
```

Figure II-1 (1 of 2)

```
c                               Loop through flags array for each odd number.
c                               The prime's index is 'i'.  (Index 1 is for
c                               the number '3'.)
          do 91 i = 1,8191
c                               Test for prime (always true for first time).
              if ( flags(i) .eq. .false. ) go to 91

c                               Yes, we have a prime, now do the algorithm
c                               that converts the prime's index to the actual
c                               value of the prime (for a 1-relative flag array,
c                               a 0-relative flag array uses: iprime=i+i+3).
              iprime = i + i + 1
c                               Get index of 1st odd multiple of the prime
c                               (3 times the prime's index plus 1 for a
c                               1-relative flag array, 3 times the prime's
c                               index plus 3 for a 0-relative flag array).
              k = i + iprime

c                               Loop to set all odd multiples of iprime to false
 20           if( k .gt. 8191 ) go to 90

                  flags( k ) = .false.
c                               Get index of next odd multiple of the prime.
                  k = k + iprime
                  go to 20

c                               Increment count of number of primes found.
 90           icnt = icnt + 1

c                               This is where iprime should be written out
c                               if we want to see what the primes are.
cccc          type *,icnt,iprime

 91       continue

 92   continue

c                               Write out the total number of primes found.
      write( 6,901 ) icnt
 901  format( 1x,i6,' primes')

      end
```

Figure II-1 (2 of 2)

calculations using division and addition and another (speed_test) to repeatedly perform calculations using a sine subroutine from a math library (the programs were named fpasub and speed_te, respectively, on the PC/AT). Both of these benchmarks were performed using all floating point formats available for each system tested. For the MicroVAX, this consisted of single-precision (4-byte) format, two double-precision (8-byte) formats, and quad-precision (16-byte) format.

It should be noted that an earlier version of the "fpatestsub" floating point benchmark did not use repetitive calls of a subroutine to perform the calculations. When this program was applied to the MicroVAX II, the language compilers used such extensive optimization techniques that the loops and unused calculation results were "optimized away".

C. Matrix Manipulation Benchmark

The purpose of this benchmark (name: run5) was to test both single-precision and double-precision floating point array computations and addressing, and nested iteration constructs. This benchmark factors a square matrix into a lower and upper triangular matrix with a Gaussian elimination technique. The actual lower and upper matrix decomposition algorithm is contained in a subprogram named "ludecm". This subprogram, along with a data output subprogram named "prnary", uses a variable-sized square matrix (2-dimensional array) with the array dimensions passed to the subprogram as arguments. This technique is not supported in the "C" language so an equivalent method of calculating and using array element addresses from subscripts was used for the "C" version of this benchmark. Results of benchmark runs showed superior performance from the "C" version, so a FORTRAN version was implemented (tstrun5.for) that used the same algorithm as the "C" language benchmark. The improved performance, as shown in the benchmark summary, indicates that this addressing algorithm is quicker in execution than using a double-subscripted array. It should also be noted that the "C" double precision version executes faster than the single precision version because "C" always internally uses double precision, thus the single precision version incurs additional time performing single-to-double and double-to-single conversions.

D. Disk Write Timing Benchmark

A test of effective I/O throughput to disk was implemented to determine the time required to write a series of large records to disk, as would be typical in a real-time data acquisition application. The timing test program (name: timtes) was designed to write 300 records of 2,048 16-bit words (4,096 bytes) per record to the disk. For the Motorola TeleSoft Ada and Motorola FORTRAN benchmark, UNIX disk limitations resulted in a maximum of only 256 records that could be written. For these cases, the time recorded in the benchmark summary was extrapolated from 250 record runs to provide an equivalent time for 300 records. In addition, a compiler error in release 3.0 of Alsys Ada for the SUN 3/260 (see a discussion of compiler problems in Section IV of this report) limited the maximum record size to only 2,047 16-bit words per record for that version of the benchmark. Therefore, times recorded for SUN Alsys Ada are based on 2,047 word records.

In addition to this benchmark series, a MicroVAX-specific disk timing analysis using VAX/VMS Record Management Services (RMS) options was also performed. This was accomplished by developing two additional versions of the disk write timing benchmark utilizing RMS system services. The first version (named timtesblok) performed block mode I/O directly to a disk file with no intermediate buffering, while the second version (named timtesblokput) used RMS "put" calls for buffered record mode I/O with large intermediate buffers. An RMS buffer count of 22 was used for this benchmark (VMS quota limits prevented greater buffering). Both versions use the "useropen" subroutine method of accessing the required RMS data structures.

E. Ada Tasking

1. Ada Rendezvous Response Benchmark

This benchmark, (name: timetask1), is an Ada two-task response time test for the purpose of determining the overhead required by the system for a rendezvous. A rendezvous is used to enable the synchronization of the two tasks in order to give an idea of how fast a task can respond to being started by another task. The design flow chart for this benchmark is presented in Figure II-2. This particular benchmark performs the rendezvous operation 1,000 times, so that the execution time listed in the benchmark summary in Section I can be divided by 1,000 to obtain the time for a single rendezvous.

2. Ada Two-Task Data Transfer Benchmark (Send/Receive)

This benchmark (name: adasend) is a send/receive data transfer test that was designed to study the throughput available in a synchronized

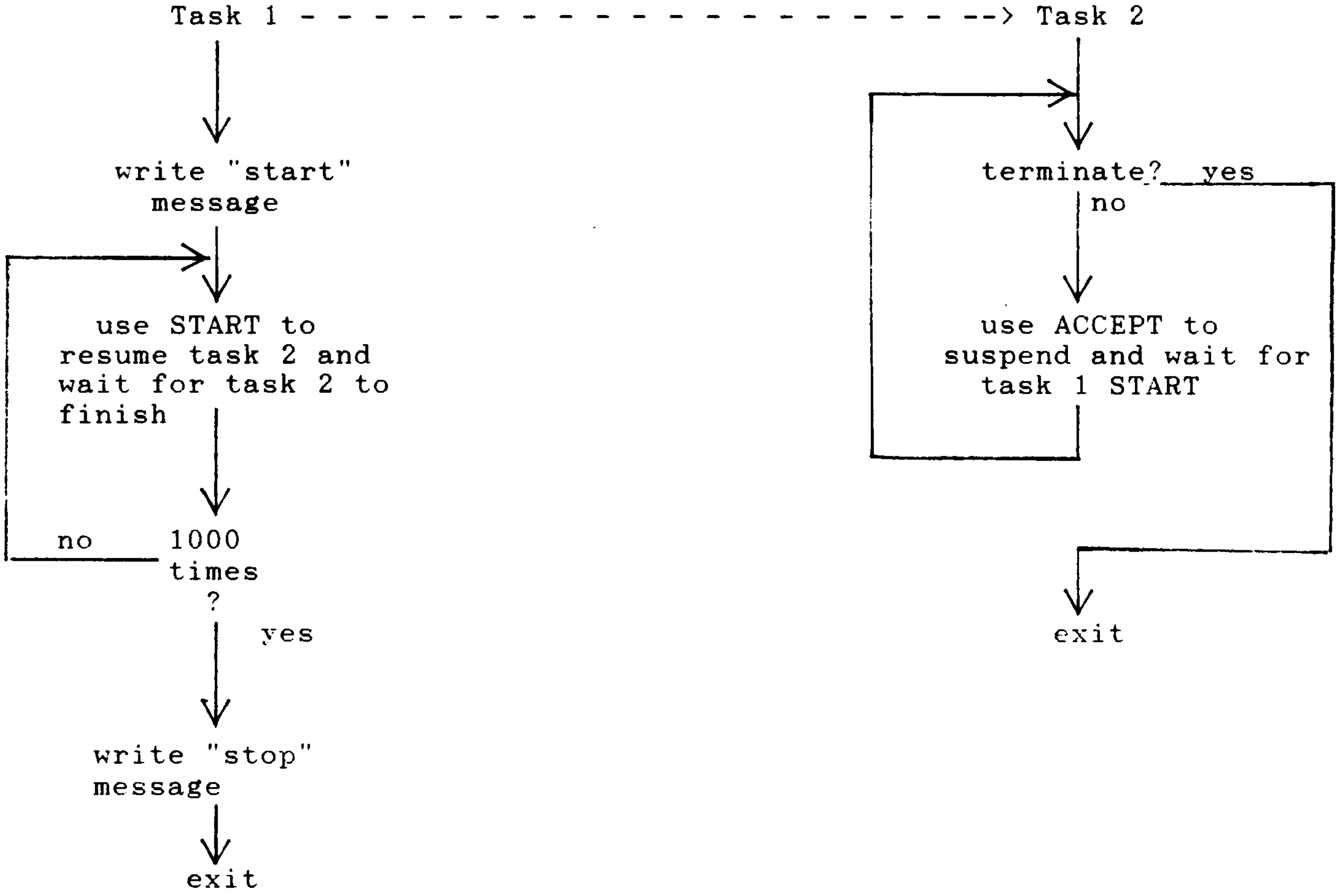

Task 1
Task 2
write "start" message
use START to resume task 2 and wait for task 2 to finish
no
1000 times ?
yes
write "stop" message
exit
terminate?
yes
no
use ACCEPT to suspend and wait for task 1 START
exit

Figure II-2 Two-Task Response Time Flowchart

multi-task environment wherein one task generates data and transfers it in memory to another task which then processes the data. The data quantity was based on previous disk transfer tests of 300 transfers of 2,048 16-bit words (4,096 bytes) each. The design flow chart for this benchmark is presented in Figure II-3. This test was used as a comparison with similar versions of the benchmark written in other languages and using system service calls to perform similar synchronized data transfer between tasks (or processes). See Section II.F.1 for a description of these system service versions. This allowed a direct comparison of execution overhead of Ada tasking services versus that of operating system services.

3. Ada Two-Task Data Transfer via Shared Disk File Benchmark

This benchmark (name: adasend_disk) is based on the two task data transfer (send/receive) benchmark but it transfers the data via a shared disk file. This benchmark allows simultaneous, asynchronous access to a disk file by the two tasks, each of which control reading and writing with shared variables, thus testing the capabilities for sharing variables as well as sharing disk files.

F. System Services

1. Send/Receive

In order to analyze the capability of an operating system to provide multiple tasks with the means of transferring data and synchronizing their access to that data, a send/receive two-task data transfer benchmark utilizing system services was developed. This benchmark is functionally identical to the Ada two-task data transfer benchmark described under Ada tasking tests, except that separate operating

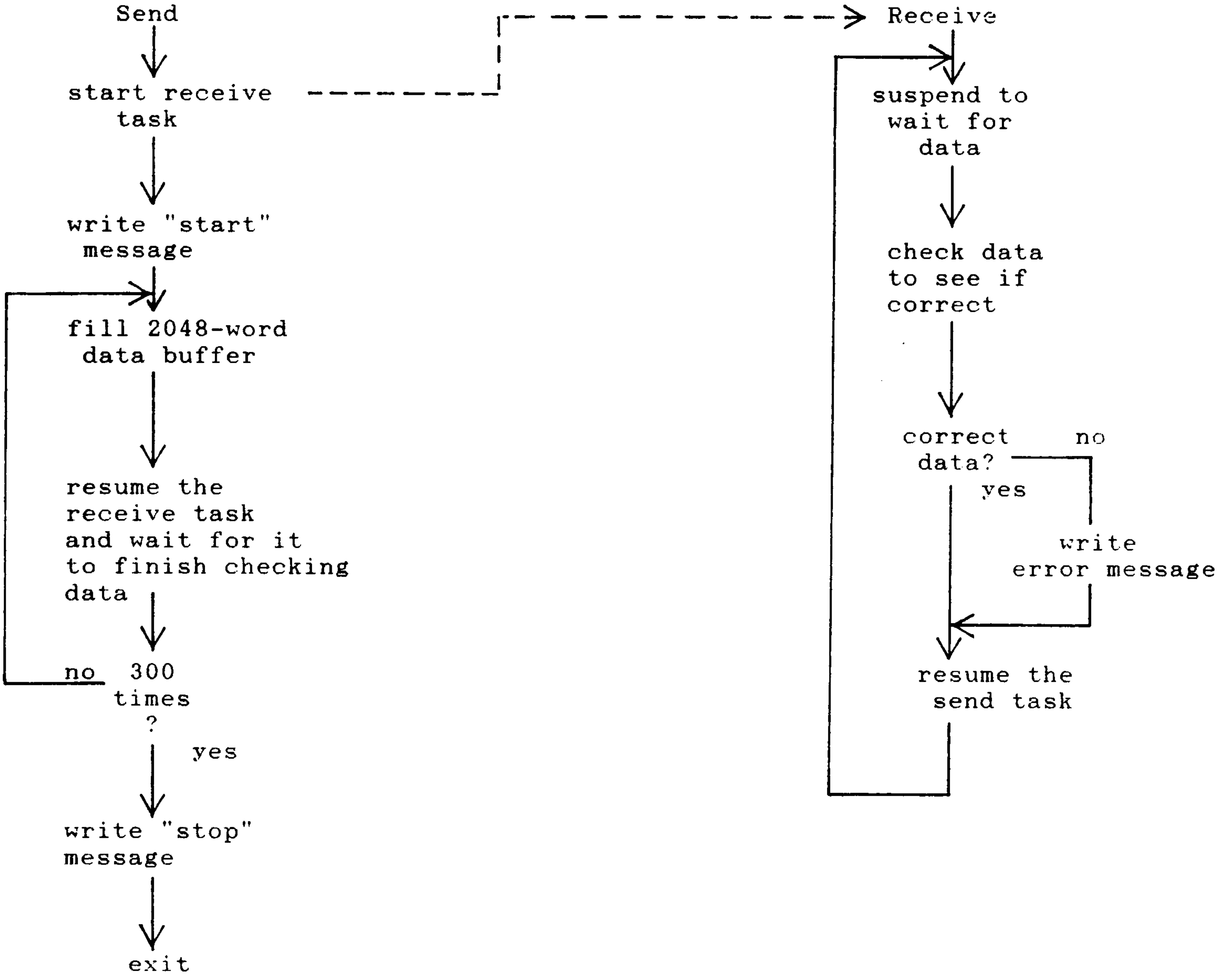


Figure II-3 Multi-task "Send/Receive" Data Transfer Design Flow Chart

system "processes" are used in place of Ada "tasks". The benchmark is designed to cause a "send" process to transfer 300 data buffers of 2,048 16-bit words each to a "receive" process, which checks the data to verify that no data was lost, insuring that synchronization was maintained. The flowchart for this benchmark is the same as for the Ada version presented in Figure II-3. For the MicroVAX II, several versions of the benchmark were developed, each testing a different method of synchronizing the two processes or testing a different method of transferring the data from one process to another. These various versions were written in VAX FORTRAN to allow easiest access to the system services and are named such that names begin with "forsend" or "forrecv" for the send and receive processes, respectively.

As a comparison, some versions were also developed for the SUN 3/260 workstation. These versions were written in "C" since that language is the "native" language of the SUN's UNIX operating system. They are named such that names begin with "csend" or crecv" for the send and receive processes, respectively.

2. Process Creation

As part of the analysis of operating system services, a study of process (or "task") creation response timing was performed by implementing a process creation benchmark. For the MicroVAX II, versions of this benchmark were developed for both subprocesses and detached processes to determine the time required to create a process. These programs were tested with various options such as with/without error checking, with installed images, using process termination mailboxes, etc., in an attempt to determine the minimum time required.

The benchmark consists of a driver process (names beginning with "rspns1") that creates a second process (named "rspns2") that does nothing but exit. The driver process waits while the 2nd process is being created, then resumes control when the 2nd process terminates. This action is repeated for 100 iterations to obtain a representative time for a single process creation cycle. As can be seen from the benchmark summary, the results of this benchmark series indicate that the version most simple to program, but slowest executing, is creation of a subprocess using the LIB$SPAWN call. The fastest executing version, but more complex to program, was creation of a detached process with the SYS$CREPRC call using a termination mailbox, where the executable image for the detached process was made an "installed" image with the Install utility. Analysis was also performed for process images defined as foreign commands but this had no affect on timing results.

3. Synchronization

Process synchronization techniques and timing results can be seen in the send/receive benchmarks described previously. However, as a direct comparison to the Ada rendezvous response time benchmark for the MicroVAX II, a program was implemented that performed a system service equivalent of the Ada rendezvous response benchmark. The benchmark used the SYS$HIBER and SYS$WAKE service calls to accomplish the synchronization, since these services were among the most time-efficient, based on the results of the send/receive tests. It consisted of two VMS processes (named timetask1.for and timetask2.for) wherein the first process "wakes" the second and "hibernates", then the 2nd process "wakes" the first and "hibernates". This action is then repeated for 1,000 iterations to provide a means of timing the synchronization. As can be seen from the benchmark data, this method was slightly faster than the Ada rendezvous time, .0012 seconds versus .0017 seconds (per synchronization cycle).

III. TEST PROGRAMS

A. Multi-Process Timing Test

The purpose of this test was to verify correct process scheduling when running multiple copies of a program. This was performed by creating a "parent-child" process (name: proctim) wherein a "parent" program created and initiated multiple copies of a "child" program. Each of the "child" programs then ran independently and periodically displayed a sequence number which identified each process as it executed. By observing this sequence number over a period of time, correct process scheduling could be verified. This test was implemented using Ada tasking for all Ada compilers, "C" on the SUN using UNIX system calls, and two versions using VAX/VMS system calls: one for subprocesses (named proctim_sub) and one for detached processes (named proctim_det). All tests showed correct execution with no time skew.

B. Ada Multi-Tasking Scenario

The purpose of this test (name: task1_and_task2) was to demonstrate multi-tasking concurrency via the concurrent execution of two Ada tasks. The first task starts the second task, then each task runs independently and asychronously while periodically writing a message to the terminal screen. A screen copy of the output from this program as executed on the MicroVAX II is shown in Figure III-1.

Ada Multi-tasking Scenario (No Synchronization)

No delays

```
$ ada task1_and_task2.ada
$ acs link task1
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run task1
 task2 starting time ( in seconds from midnight ) =   50451.5600
 task2 running, time ( in seconds from midnight ) =   50451.6600
 task2 running, time ( in seconds from midnight ) =   50451.7400
 task2 running, time ( in seconds from midnight ) =   50451.8100
 task1 starting time ( in seconds from midnight ) =   50451.8900
 task1 running, time ( in seconds from midnight ) =   50451.9800
 task1 running, time ( in seconds from midnight ) =   50452.0500
 task1 running, time ( in seconds from midnight ) =   50452.2000
 task1 running, time ( in seconds from midnight ) =   50452.3600
$
$
$
$
```

One 5 sec. delay in Task2

```
$ ada task1_and_task2.ada
$ acs link task1
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run task1
 task2 starting time ( in seconds from midnight ) =   51716.5900
 task2 running, time ( in seconds from midnight ) =   51716.7000
 task2 running, time ( in seconds from midnight ) =   51716.7700
 task1 starting time ( in seconds from midnight ) =   51716.8400
 task1 running, time ( in seconds from midnight ) =   51716.9200
 task1 running, time ( in seconds from midnight ) =   51716.9900
 task1 running, time ( in seconds from midnight ) =   51717.2000
 task1 running, time ( in seconds from midnight ) =   51717.2700
 task2 running, time ( in seconds from midnight ) =   51721.8500
$
$
$
$
$
$
```

5 second delays in both tasks

```
$ ada task1_and_task2.ada
$ acs link task1
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run task1
 task2 starting time ( in seconds from midnight ) =   49309.2900
 task1 starting time ( in seconds from midnight ) =   49309.4000
 task2 running, time ( in seconds from midnight ) =   49314.4100
 task1 running, time ( in seconds from midnight ) =   49314.4800
 task2 running, time ( in seconds from midnight ) =   49319.4900
 task1 running, time ( in seconds from midnight ) =   49319.5600
 task2 running, time ( in seconds from midnight ) =   49324.5700
 task1 running, time ( in seconds from midnight ) =   49324.6400
 task1 running, time ( in seconds from midnight ) =   49329.7300
$
$
$
$
$
$
```

Figure III-1

C. VAX High-Level Language Access to System Services

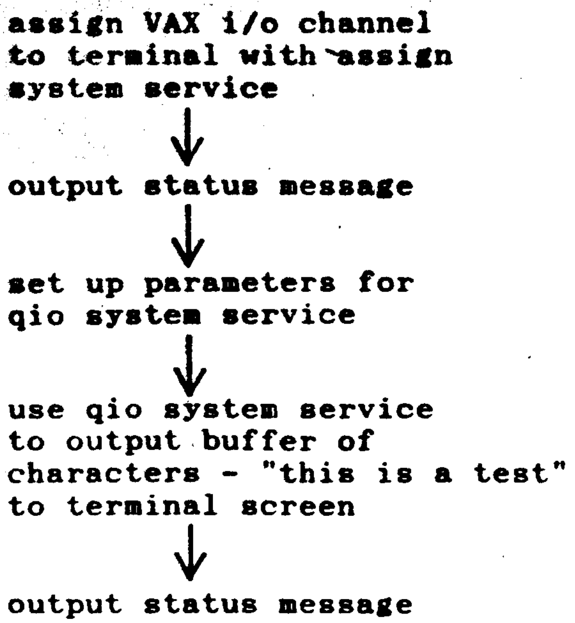
This test was used to demonstrate the VMS operating system's capability for high-level language access to the system services. To accomplish this, a test program implementing the Queued Input/Output system service (QIO) was implemented on the MicroVAX II. This program was written in VAX Ada (ttqio.ada), VAX FORTRAN (ttqio.for), and in "C" (ttqio.c). The Ada program used the VAX Ada STARLET package which is provided for access to system services from Ada.

The system services necessary for access to an operating system's I/O devices directly through its I/O drivers are an important consideration for real-time applications and this test successfully demonstrated this capability. The design flow chart for this benchmark is presented in Figure III-2. The screen copy of the output from the Ada and "C" versions is also presented in Figure III-2.

D. VAX Alarm Test

The VAX Alarm Test was a series of programs developed to test the MicroVAX VMS timer system calls. Two basic versions of this alarm test program were implemented. One version (testalarm) causes a system event flag to be set after a specified time period. The other version (testalarm_ast) causes an AST routine to be executed after a specified time period. This version was implemented in FORTRAN (testalarm_ast.for), in "C" (testalarm_ast.c), and in Ada (testalarm_ast.ada). The implementations of the "AST" version verified that AST routines could be utilized from Ada and "C" programs as well as from FORTRAN.

VAX qio system service access



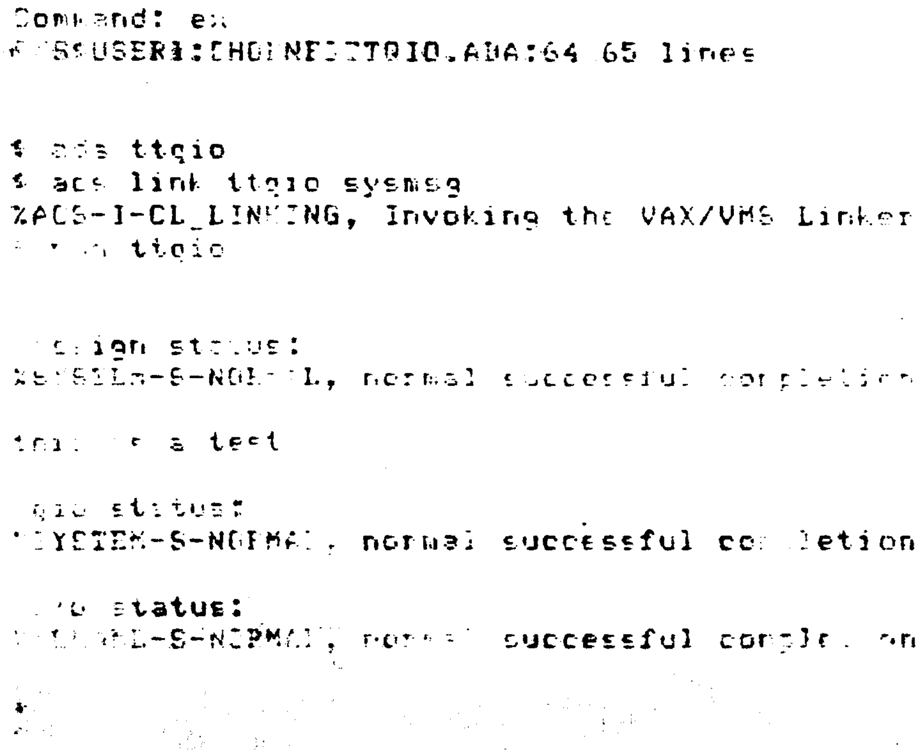
assign VAX i/o channel to terminal with assign system service

output status message

set up parameters for qio system service

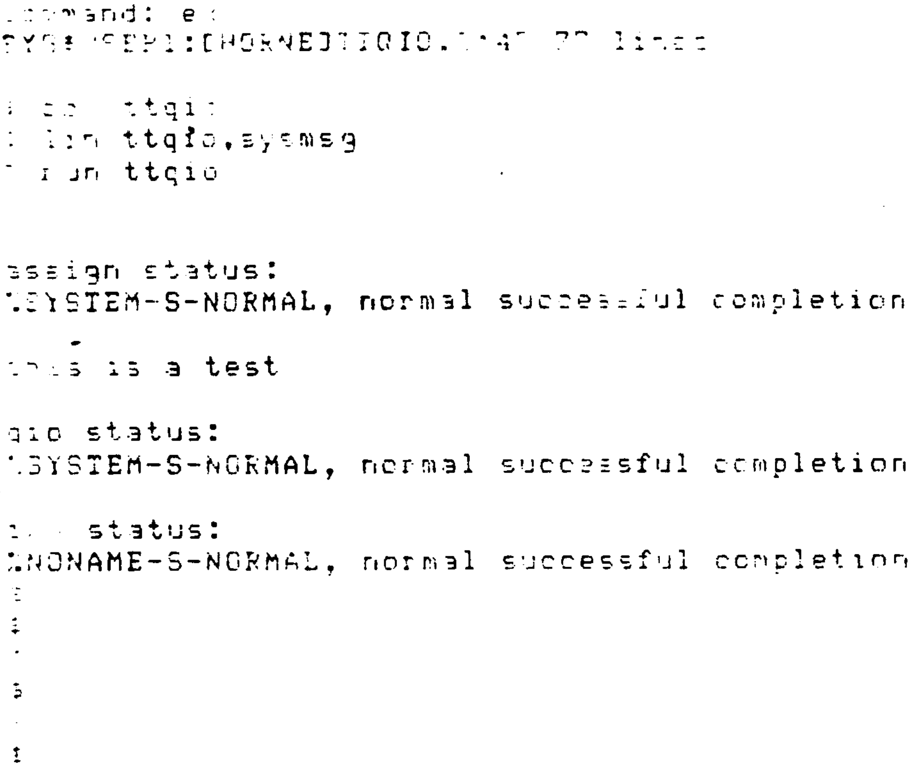
use qio system service to output buffer of characters - "this is a test" to terminal screen

output status message



```
Command: ex
$USER1:[HOINF]TTQIO.ADA;64 65 lines


$ acs ttqio
$ acs link ttqio sysmsg
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run ttqio


assign status:
%SYSTEM-S-NORMAL, normal successful completion

this is a test

qio status:
%SYSTEM-S-NORMAL, normal successful completion

i/o status:
%SYSTEM-S-NORMAL, normal successful completion

$
```

Figure III-2 (1 of 2)



```
command: ex
SYS$USER1:[HORNE]TTQIO.[illegible]47 77 lines

$ mac  ttqio
$ lin ttqio,sysmsg
$ run ttqio


assign status:
%SYSTEM-S-NORMAL, normal successful completion

this is a test

qio status:
%SYSTEM-S-NORMAL, normal successful completion

iosb status:
%NONAME-S-NORMAL, normal successful completion
$
$

$

$
```

Figure III-2 (2 of 2)

E. Ada "delay" Test

The purpose of this test was to simulate periodic processing at a fixed time interval in Ada. This was accomplished by implementing a program (name: adadelay) to use the Ada "delay" statement and the "calendar" package. Results of this test showed accurate timing for the MicroVAX system, but inaccurate timing for the Motorola VM03 system. This was determined to be because the TeleSoft Ada compiler does not use the system clock for the delay, but instead uses simulated timing via software. The Alsys Ada compilers also produced programs that executed with accurate timing.

F. Ada Task Order Test

The Ada Task Order test was used to determine the order of task activation when multiple task declarations were made in Ada. This test was devised such that four independent tasks were declared in an Ada main program (name: task_order), then each task would identify itself on the operator's terminal when it was activated. Results from this test showed that for all Ada compilers tested except VAX Ada, the tasks were activated in the order declared. For the MicroVAX, tasks were activated in the reverse order of their declaration.

G. Multi-Task Time-shared Execution Test

This test was developed to determine if there were deficiencies in a real-time Ada application where one task must control the execution of another task. This test (name: task_exec, for the PC/AT: tsk_exec) was devised where a master task attempts to schedule execution of 3 sub-tasks such that each sub-task executes for a specified time interval (5 seconds in the test case). The master task accomplishes this with a reentrant "check" subprogram for checking execution time. This test was run on the MicroVAX II, the SUN 3/260, and the IBM PC/AT. Results of this test for the MicroVAX showed an actual allocation of 5 seconds of execution time for each task, using a 5 second delay in the check subprogram. For the PC/AT, execution time was 5.0 to 5.1 seconds, while for the SUN, execution time was 5.0 to 5.2 seconds. In addition, some anomalies were encountered during the execution of this test for the SUN. These were: 1) Task 2 and Task 3 of the 3-task set would terminate if screen output was held; 2) the test would only run in non-window mode, it would not run with windows (SUNTOOLS) active.

Also, as a comparison, a version was developed for the PC/AT using the Alsys Ada compiler's "time-slice" option. This version (name: tstslice) resulted in an execution allocation of varying amounts from 5.6 seconds to 13.9 seconds for the test case timed. Since the time-slice feature was not available for SUN Alsys Ada, this test could not be run on that system.

IV. SYSTEM AND COMPILER ANALYSIS

A. Ada Terminal Input Analysis

Some problems developing Ada programs that performed input from the terminal keyboard led to an investigation of the details of Ada terminal input on the MicroVAX and SUN systems. These problems occurred when using the various forms of Ada GET and GET_LINE procedures in certain combinations. The Ada Language Reference Manual and other reference material contain little information on this subject, therefore a series of experiments were performed to determine the behavior of these procedures for each of the data types. It was found that using GET for numeric (integer, float, etc.) or enumerated types returned the valid input, but left the line terminator as the next character in the "read" buffer, so that if a GET_LINE of a string followed, the terminator caused the input to immediately complete with no characters read. It is necessary to use a SKIP_LINE call after the GET to bypass the line terminator. The same problem occurs if invalid input is entered. The invalid input characters remain in the "read" buffer after the exception generated by the invalid input, and must be bypassed with a SKIP_LINE call to prevent re-reading this invalid input during a retry attempt. An example of the lack of information in this area is the fact that Example 14.7 of the Ada Language Reference Manual has such an error in its exception handling code. This example neglects to include a SKIP_LINE call after a data error. If this example is implemented as written, an endless loop will occur after a data error generated by the input of leading non-alpha characters. A corrected version of this example is shown in Figure IV-1.

```
-- dialogue.ada  -  This is the example program given in 14.7 of the Ada LRM.

with text_io;  use text_io;
procedure dialogue is
    type color is (white, red, orange, yellow, green, blue, brown);
    package color_io is new enumeration_io(enum => color);
    package number_io is new integer_io(integer);
    use color_io, number_io;

    inventory : array (color) of integer := (20, 17, 43, 10, 28, 173, 87);
    choice : color;

    procedure enter_color (selection : out color) is
    begin
        loop
            begin
                put ("color selected: ");               -- prompts user
                get (selection);                        -- accepts color typed, or
                                                        -- raises exception
                return;
            exception
                when data_error =>
                    put("invalid color, try again.  ");
                    new_line(2);

--*****************************************************************************
--  Note: The following line must be added to the example given in 14.7 of
--        the Ada LRM in order to correctly handle data errors generated by
--        the input of leading non-alpha characters.

                    skip_line;
--*****************************************************************************

                    -- completes execution of the block statement
            end;
        end loop;   -- repeats the block statement until color accepted
    end;
begin                                                   -- statements of dialogue

    number_io.default_width := 5;

    loop

        enter_color(choice);                            -- user types color and new line

        set_col(5);  put(choice);  put(" items available:");
        set_col(40);  put(inventory(choice));   -- default width is 5
        new_line;
    end loop;
end dialogue;
```

Figure IV-1

B. Foreign Routine Capability

The ability to call "foreign" routines from within MicroVAX Ada and to call Ada routines from main programs in other VAX languages was demonstrated by implementing VAX software to call both a FORTRAN subroutine (innerprod1.for) and a FORTRAN function (innerprod.for) from an Ada main program (main1.ada and main.ada respectively), and to call Ada functions from FORTRAN main programs (formain.for). Also implemented was an Ada main program (main2.ada) that calls a FORTRAN subroutine (sysmsg.for) which in turn calls a VAX system routine. A VAX terminal screen copy of the results of the above is shown in Figure IV-2.

The ability to call foreign routines from within Alsys Ada on the SUN was demonstrated by implementing test cases similar to those used on the DEC MicroVAX. The following cases were demonstrated:

- Calling a FORTRAN subroutine (innerprod1.f) from an Ada procedure (main1.ada).
- Calling a FORTRAN function (innerprod.f) from an Ada procedure (main2.ada).
- Calling a UNIX system routine (fork) from an Ada procedure (adafork.ada).

It should be noted that to accomplish the above mentioned cases, special Alsys Ada "binder" options were required in addition to the Ada source code pragmas. These binder options are documentated in the appropriate source code listings.

CALLING FORTRAN SUBROUTINE FROM ADA
(Returns answer as calling argument)

```
$
$
$
$
$ ada mainl
$ for innerprodl
$ acs link mainl innerprodl
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run mainl
returned value =  1.00000E+02
$
$
$
$
$
$
$
$
$
$
```

CALLING FORTRAN FUNCTION FROM ADA
(Returns answer as function value)

```
$ ada main
$ for innerprod
$ acs link main innerprod
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run main
returned value =  1.00000E+02
$
```

ORIGINAL PAGE IS
OF POOR QUALITY

CALLING ADA SUBROUTINE FROM FORTRAN

```
$
$
$
$
$
$
$ for formain
$ ada nfind
$ acs link/nomain nfind formain              ← using ACS LINK
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run nfind
          5
$
$
$ link formain,[horne.adalib]nfind           ← using DCL LINK
$ run formain
          5
$
$
$
$
$
$
$
```

Figure IV-2 (1 of 2)

```
$
$
$
```

EXAMPLE OF USING ADA TO CALL FORTRAN SUBROUTINE
THAT CALLS A SYSTEM SERVICE

```
$
$ ada main2
$ for sysmsg
$ acs link main2 sysmsg
%ACS-I-CL_LINKING, Invoking the VAX/VMS Linker
$ run main2
 Please enter system error number: 10820

%SYSTEM-F-EXENQLM, exceeded enqueue quota
$
$ run main2
 Please enter system error number: -1

%NONAME-?-NOMSG, Message number FFFFFFFF
$
$ run main2
 Please enter system error number: 1

%SYSTEM-S-NORMAL, normal successful completion
$
$
```

Figure IV-2 (2 of 2)

C. I/O Loading Analysis

The effects of the "priority" pragma and the "time slice" pragma on the order and frequency of the I/O from I/O-bound Ada tasks was studied. This was implemented by having an Ada main program (char.ada) declare two separate tasks (tasks pra and prb) which would each output a single character to the terminal screen. For a proper demonstration of the test, these two tasks need to alternate execution. This test was implemented on the MicroVAX II, the SUN 3/260 and the IBM PC/AT. The test was first run without any pragmas, in which case, the first task to be activated would output to the screen continuously without ever allowing the other task to run. This result was the same on all three systems. Then the test was run with the "priority" pragma. Again, only the first task to be activated would run, but in this case, the "priority" pragma allows the programmer to select the task to be activated first, i.e., the task with the higher priority would be activated first. This result was also the same on all systems. Finally the test was run with the "time slice" pragma. This allowed the output to the terminal to be alternated. On the PC/AT the two tasks alternated execution approximately in accordance to the time slice value. When run on the MicroVAX, the execution was alternated on a byte by byte basis, i.e., the first task would output one character, then the other task would output one character, then the first task would output again, etc. Since the "time slice" pragma is not yet implemented on the SUN 3/260 system, this option could not be tested on that system.

D. Multiple Process Loading Analysis

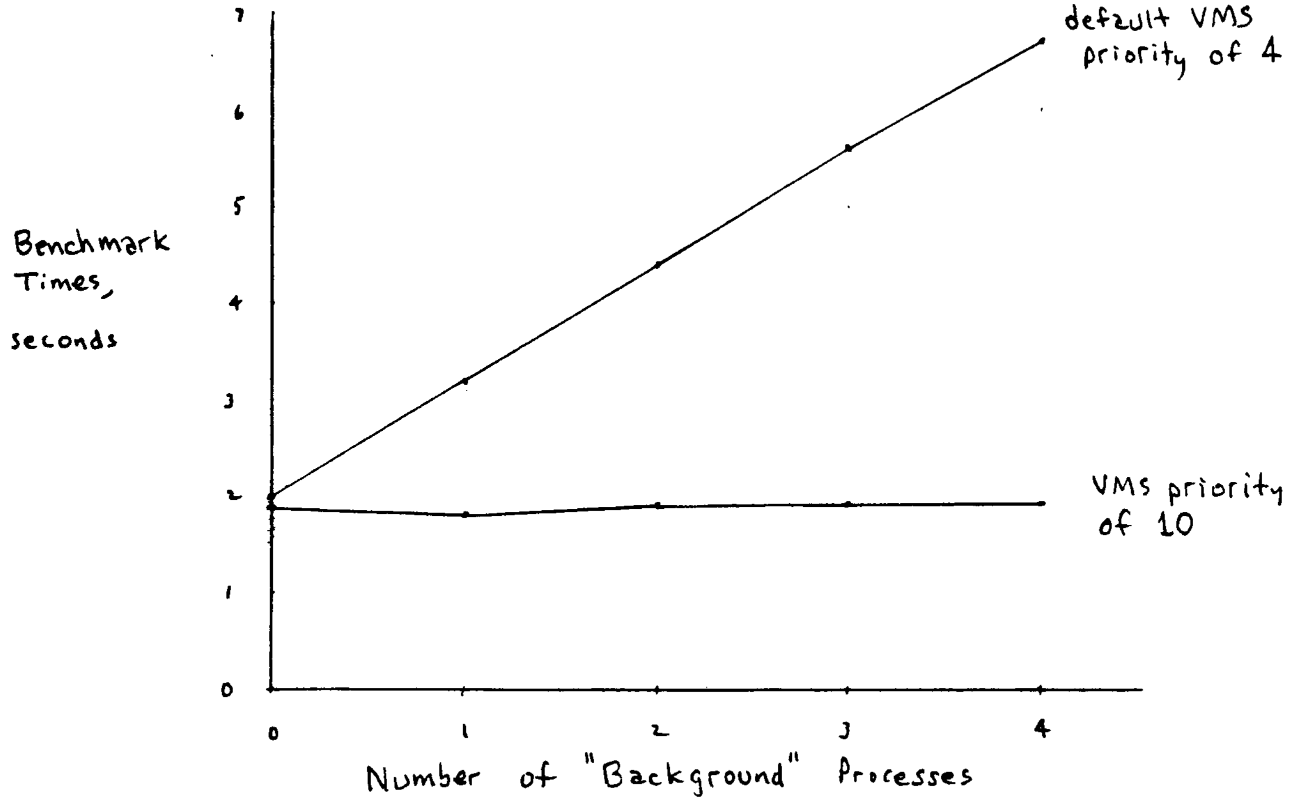
An analysis of the effects of loading and of different process priority on compute-bound processes for the DEC MicroVAX II and the SUN 3/260 was performed. This was accomplished by timing the execution of the Ada prime number benchmark while varying the number of computational processes running in the "background". The series was repeated with the benchmark running at a high operating system priority level. Graphs of the loading degradation are shown in Figures IV-3 and IV-4 for the MicroVAX and the SUN, respectively. This analysis was performed on the SUN both with and without the SUN Window environment present. There was no detectable difference in the time.

Also, an investigation of the loading effects of the SUN window environment using the SUN Ada 2-task data transfer benchmark. was performed. No degradation was observed for 1,2 or 3 windows (with no processes active in the window).

E. Large Array Analysis

This analysis studied the MicroVMS operating systems support of very large data arrays, both for size limitations and for loading degradation, using Ada and FORTRAN. A graphic representation of system loading (execution time) versus array size was generated that showed points of discontinuity where loading increased dramatically for a small increase in array size. This data is presented in Figures IV-5 and IV-6. The large size of the executable image file for the Ada version of this test was also noted.

MULTIPLE PROCESS LOADING ON MICROVAX
(Computational Processes)

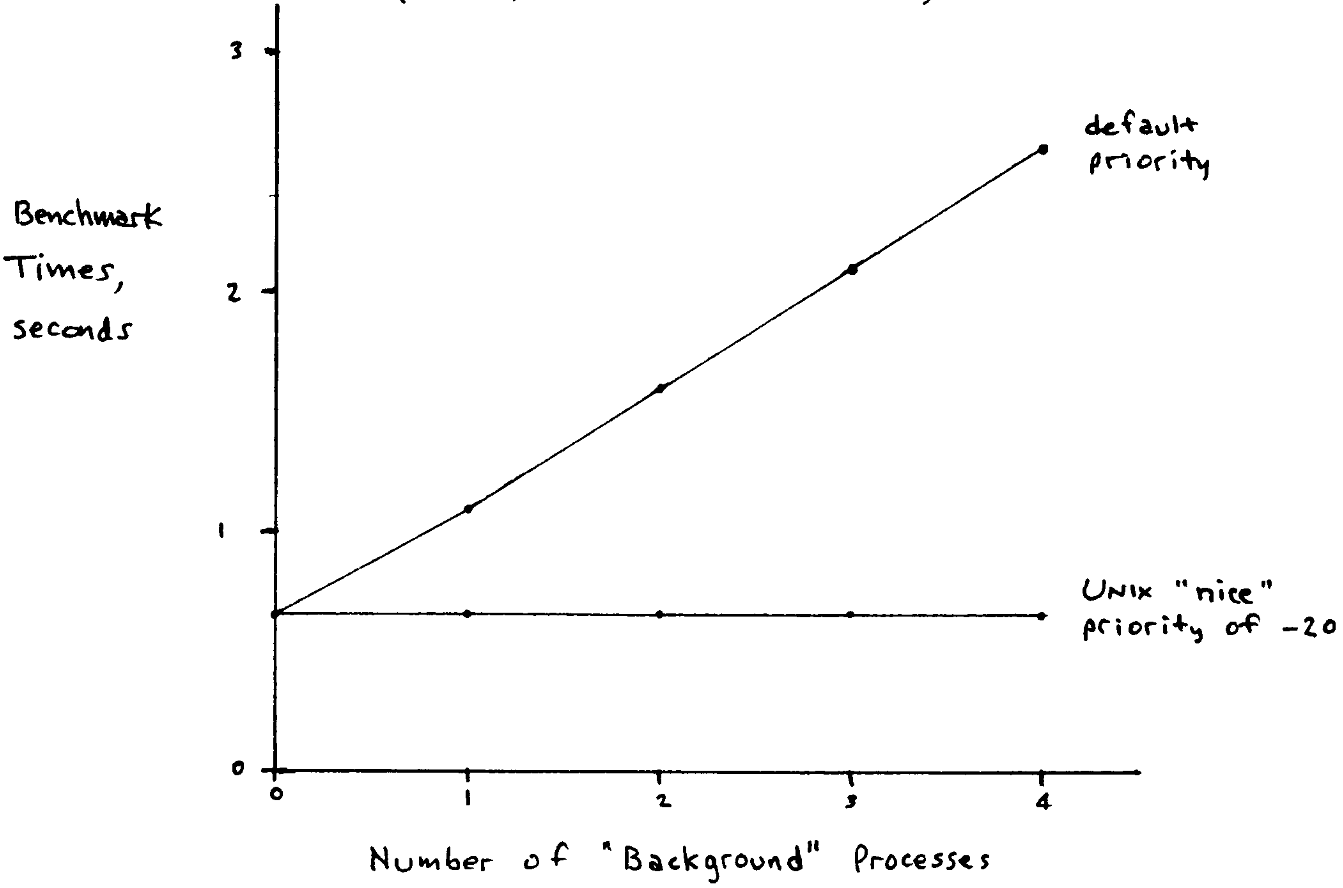


Benchmark used was Ada prime number program with suppress_all pragma.

"Background" processes were FORTRAN versions of prime number program with increased iteration count.

Figure IV-3

MULTIPLE PROCESS LOADING ON SUN 3/260
(computational Processes)



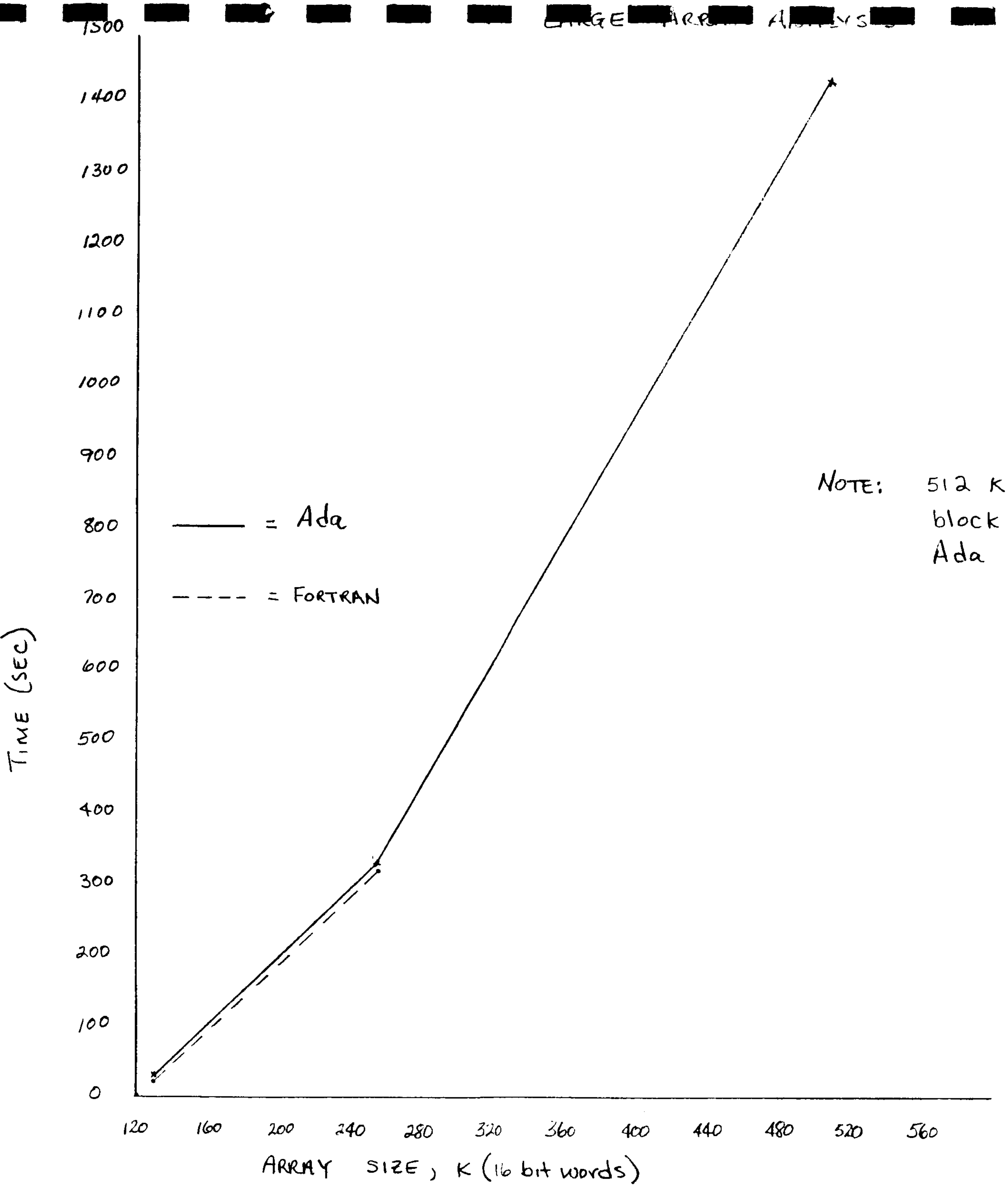
Notes:

Benchmark used was Ada prime number program (ALSYS ADA) with "checks= stack" option.

"Background" processes were FORTRAN versions of prime number program with increased iteration count.

This test was run without the SUN window environment.

Figure IV-4



NOTE: 512 K array resulted in 2055 block executable image file for Ada vs. 5 blocks for FORTRAN.

Figure IV-5

LARGE ARRAY ANALYSIS

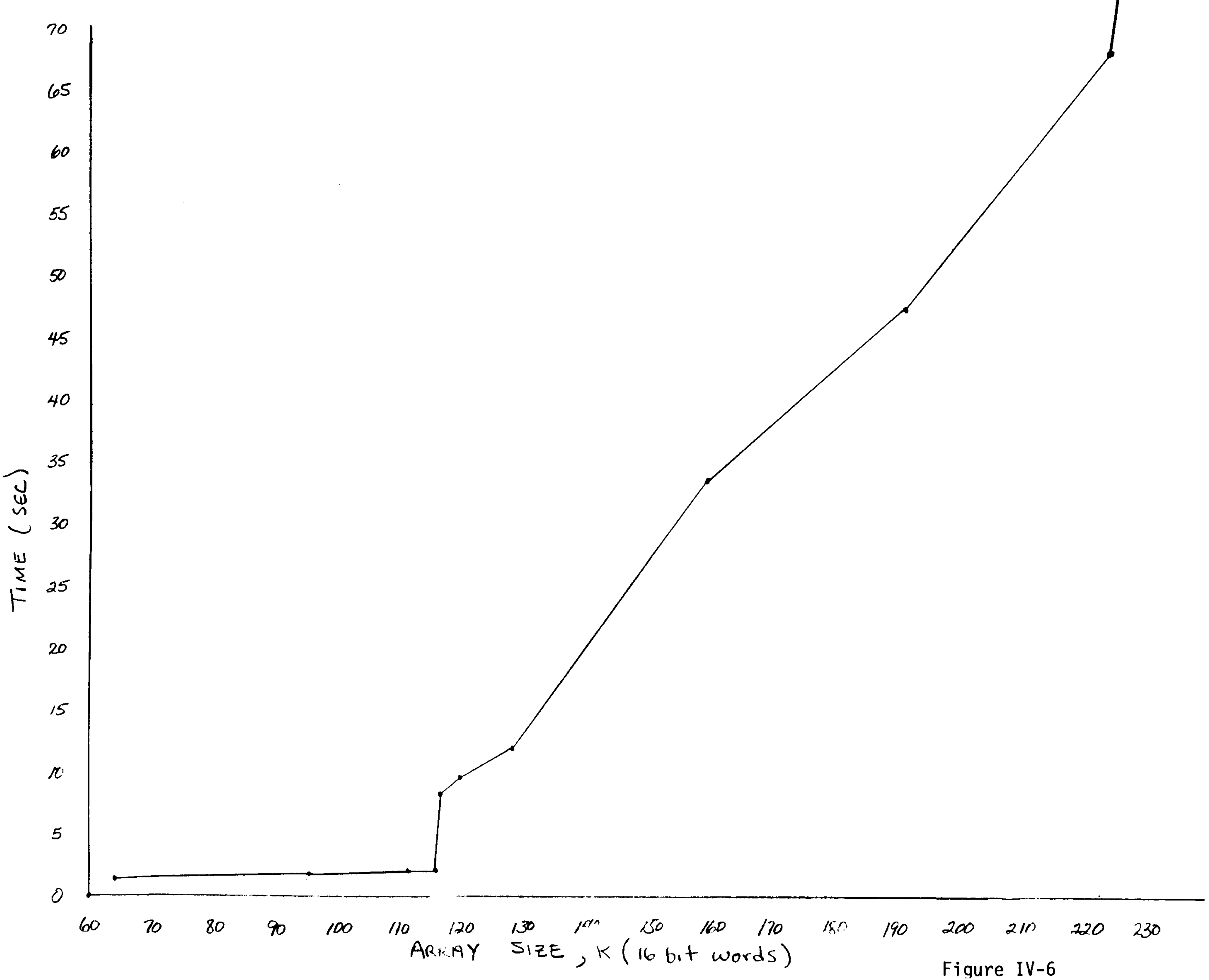


Figure IV-6

F. Ada Multiple Periodic Tasks with Calculation Task

To resolve questions concerning CPU usage during multi-tasking, an Ada test program (procload.ada) consisting of multiple periodic tasks interrupted by a calculation-intensive task was implemented. This program (procload.ada) was constructed by inserting the code of the prime number benchmark program (zprime.ada) into the body of the multiple process timing test program (proctim.ada). The purpose of this analysis was to determine if the time-slice mechanism would allow the calculations of the prime number program to be performed during the delays (idle time) of the multiple periodic tasks program. The results of this analysis are presented as follows:

Ada Test Scenario - Multiple Periodic Tasks with Calculation Task

Alsys Ada - PC/AT

with timeslice=10 msec	-	322 sec.
without timeslice	-	507 sec., tasks out of order after calculation task runs
calculation task alone	-	213 sec.

MicroVAX Ada

with timeslice=10 msec	-	322 sec.
without timeslice	-	355 sec., tasks stayed in order after calculation task runs
calculation task alone	-	55 sec.

This shows the need for a time-slice scheduling mechanism for all Ada compilation systems.

G. VAX/VMS Ada Analysis

While performing the benchmark study, it was found that the VMS Ada compiler defaults to the g_float format when performing calculations involving 64-bit (long_float) real numbers. However, if this is changed with "pragma long_float (d_float)", the system then uses the d_float format from that point on until it is changed back with "pragma long_float (g_float)". It should also be noted that when using the g_float format in conjunction with the math library this format is implemented with software rather than hardware and therefore executes much more slowly. Also, it was discovered that d_float is incompatible with the VAX predefined instantiation of the long_float math library (long_float_math_lib). This predefined instantiation of the library is expecting g_float, therefore, in order to use d_float a user instantiation of the library must be made.

While performing the I/O loading analysis (see Section IV.C) it was noted that the "priority" pragma and the "time slice" pragma exhibited behavior that was not expected. The VAX allows priority values in the range from 0 to 15 with a default priority value of 7. The data gathered during the I/O loading analysis (char.ada) indicates that although the VAX allows a priority value range of 0-15, the priority value chosen has no bearing on the actual scheduling of the tasks. It appears that any priority values above 7 (8-15) or any values below 7 (0-6) are treated as equal values, i.e., a priority of 14 is the same as a priority of 8 or a priority of 6 is the same as a priority of 0. A task with the default priority of 7 will be scheduled before any task with any priority less than 7 and after any task with any priority greater than 7.

It was observed that the VAX time-slice pragma apparently causes round robin scheduling to occur once the time specified in the pragma has elapsed. This observation was based on the execution of "char.ada" which has two Ada tasks, each of which have an infinite loop in which a single character is output to the terminal screen. Without any pragmas, the first task to run will run continuously without allowing the second task to execute. With the time-slice pragma set to 5 seconds, the first task to execute will run for approximately 5 seconds, then the two tasks will alternate execution with each task transmitting a single character to the terminal screen. It would seem that each task should run for 5 seconds and then release the CPU to the other task, instead of the much faster scheduling behavior actually observed.

H. Alsys Ada (SUN) Analysis

During the performance of the benchmark study for the SUN workstation, two Alsys Ada compiler anomalies were encountered. These two problems were observed while implementing the disk write timing benchmark (timtes) and the matrix manipulation benchmark (run5). The error that occurred with the disk write timing benchmark was that the maximum record size that could be successfully compiled was only 2,047 16-bit words. Any larger record size (this benchmark was designed for 2048-word records) would generate a fatal compiler error. For the matrix manipulation benchmark, a run-time error would be generated when attempting to execute the long_float version of the program. Test cases with minimum code were developed to demonstrate the two anomalies, and these two test cases were presented to Alsys, Inc. who acknowledged that they were indeed compiler errors. According to Alsys, both are to be corrected in later compiler releases. The test programs to demonstrate the errors that occurred in the disk write timing benchmark and the matrix manipulation benchmark are presented in Figures IV-7 and IV-8, respectively. (The program names are test1.ada and test2.ada, respectively). Figure IV-9 presents an intermediate step that is a "work around" to the problem that occurred in the matrix manipulation benchmark.

In addition to the analysis of the compiler anomalies for the SUN Alsys Ada compiler, an analysis of several compiler/binder options were performed on a representative set of the benchmark programs for the SUN. Table IV-1 presents the results of this analysis.

It should be noted that the Alsys Ada compiler for the SUN does not provide any form of a time-slice mechanism.

I. Alsys Ada (PC/AT) Analysis

During the course of the study, release 3.2 of the Alsys Ada compiler for the PC/AT was received and analysis performed. This version (3.2) of the compiler for the IBM PC/AT contains a math library, but this math library is user supplied and <u>is</u> <u>not</u> <u>supported</u> by Alsys Inc. To access this library the user must perform the following steps:

```
-- test1.ada  -  SUN 3/260 version. ALSYS Ada compiler. Demonstrates the
--                compiler error found in timtes.ada.

with sequential_io;

procedure test1 is

    type buffer is array( integer range 1..2048 ) of integer;

    package buffer_io is new sequential_io( buffer );
    use buffer_io;

    data_file : buffer_io.file_type;

begin

    create( file => data_file,            -- create a file.
            name => "timetest.dat" );

end test1;
```

Figure IV-7

```
-- test2.ada  --  SUN 3/260 version. ALSYS Ada compiler. Demonstrates the
--                 "PROGRAM_ERROR" found in run5.ada.

procedure test2 is

    type buf is array( integer range 1..2,integer range 1..2 ) of long_float;

    b      : buf := (( 1.0,1.0 ),
                     ( 2.0,3.0 ));

    mult   : long_float;
    diag   : integer := 1;
    row    : integer := 2;

begin                                     -- beginning of test2.

    mult := b( diag,row ) / b( diag,diag );

end test2;                                -- end of test2.
```

Figure IV-8

```
-- test3.ada  --  SUN 3/260 version. ALSYS Ada compiler. Demonstrates the
--                "work-around" for the "PROGRAM_ERROR" found in run5.ada.

procedure test3 is

    type buf is array( integer range 1..2,integer range 1..2 ) of long_float;

    b     : buf := (( 1.0,1.0 ),
                    ( 2.0,3.0 ));

    mult  : long_float;
    diag  : integer := 1;
    row   : integer := 2;
    temp1 : long_float;
    temp2 : long_float;

begin                                   -- beginning of test3.

    temp1 := b( diag,row );
    temp2 := b( diag,diag );
    mult := temp1 / temp2;

end test3;                              -- end of test3.
```

Figure IV-9

SUN Alsys Ada benchmark analysis (Execution Time in Seconds)

			Improve=>		checks=>
	All	68881	(reduction	checks=>	stack
Benchmark	Defaults	option	=>extensive)	stack	w/68881
Sieve Prime Number	.89	-	.84	.66	-
Floating point, sine function					
float	4.4	4.0	4.0	3.9	3.9
long-float	9.6	7.0	9.4	8.8	6.3
Floating point, subroutine					
(no math lib.) float	6.8	6.8	6.8	7.4	7.2
long-float	10.0	8.0	10.0	10.8	8.9
Matrix					
float	13.9	13.3	12.5	12.3	11.6
long-float	15.8	16.3	14.4	13.9	14.4
2-task data transfer	3.2	-	2.7	2.6	-

Table IV-1

1. copy the math library package specification (mathl.ads) and the math library package body (mathl.adb) into the user directory. (note: these reside in \alsys\math)

2. compile both of the above into the user library, and

3. use the "bind" option interface=(search=\alsys\math\math.lib) when building the executable.

Additional study of the PC/AT Alsys Ada compiler consisted of an analysis of several compiler/binder options that were performed on a representive set of the benchmark programs for the PC/AT. The results of this analysis is presented in Table IV-2. This table also presents the times obtained using the previous version (3.1) of the Alsys Ada compiler. These times were obtained using all default values.

PC/AT Alsys Ada 3.2 Benchmark Analysis (Execution Time in Seconds)

Benchmark	All Defaults	Tasking =No	Checks =Stack	Reduction = Partial	Previous Version
Sieve Prime Number, execute	5.1	5.1	3.7	4.9	6.2
compile, bind	101.0	-	-	-	89.1
Floating point, sine function					
float	8.7	8.7	8.7	8.7	N/A
long-float	9.9	9.9	9.9	9.9	N/A
Floating point, subroutine					
(no math lib.) float	27.7	27.7	27.7	27.6	28.0
long-float	45.5	45.5	45.4	45.3	N/A
Matrix					
float	71.2	71.2	61.0	62.8	76.9
long-float	76.8	76.9	67.0	68.8	N/A
2-task data transfer	19.1	N/A	16.9	16.9	29.2
Rendezvous response	1.0	N/A	1.0	1.0	1.0
Disk write timing	13.4	13.4	13.4	13.4	15.5

Table IV-2

V. SOURCE LISTINGS

This section contains program source file listings of the software developed during the course of the study. Unless otherwise noted in the comments at the beginning of the program or in the following list, the programs listed are the versions of benchmarks and test programs that were developed for, and executed on, the DEC MicroVAX II system. The following source listings are included in this section:

1. zprime.ada
2. zprime.for
3. zprime.c
4. zprime.mar
5. zprime.p (SUN)
6. zprime.s (SUN)
7. fpatestsub.ada
8. fpatestsub.for
9. fpatestsub.c
10. speed_test.ada
11. speed_testsav.ada (long_float)
12. speed_test.for
13. speed_test.c
14. run5.ada & subprograms ludecm & prnary
15. run5.for & subroutines ludecm & prnary
16. run5.c & functions ludecm & prnary
17. tstrun5.for & subroutines ludecum & prnary
18. timtes.ada
19. timtes.for
20. timtes.c
21. timtes.p (SUN)
22. timtesblok.for
23. timtesblokput.for
24. timetask1.ada
25. adasend.ada
26. adasend_disk.ada
27. forsend.for/forrecv.for
28. forsendsave1fil.for/forrecvsave1.for
29. forsendsave6.for/forrecvsave6.for
30. csend1.c/crecv1.c (SUN)
31. csend2.c/crecv2.c (SUN)
32. csend3.c/crecv3.c (SUN)
33. rspns1save1.for/rspns2.for
34. rspns1save5qio.for
35. timetask1.for/timetask2.for
36. proctim.c (SUN)
37. proctim_sub.for
38. proctim_det.for
39. proctim1.for

40. proctim.ada
41. task1_and_task2.ada
42. ttqio.ada
43. ttqio.for
44. ttqio.c
45. testalarm.for
46. testalarm_ast.for
47. testalarm_ast.ada
48. testalarm_ast.c
49. adadelay.ada
50. task_order.ada
51. task_exec.ada
52. tstslice.ada (PC/AT)
53. main.ada
54. innerprod.for
55. main1.ada
56. innerprod1.for
57. formain.for
58. nfind.ada
59. main2.ada
60. sysmsg.for
61. main1.ada (SUN)
62. innerprod1.f (SUN)
63. main2.ada (SUN)
64. innerprod.f (SUN)
65. adafork.ada (SUN)
66. char.ada
67. procload.ada

```
--  zprime.ada  Eratosthenes Sieve Prime Number Program in VAX Ada.

with text_io;
with integer_text_io;
use text_io,integer_text_io;

procedure zprime is

    flags:array (integer range 0..8190) of boolean;
    i,prime,k,count,iter:integer;

--    pragma optimize( time );

begin

    put("10 iterations"); new_line;

    for iter in 1..10 loop
        count := 0;

        flags := ( 0..8190 => TRUE );

        for i in 0..8190 loop
            if flags(i) then
                prime := i + i + 3;
                k := i + prime;

                while k <= 8190 loop
                    flags(k) := FALSE;
                    k := k + prime;
                end loop;

                count := count + 1;
--              put(count);put(prime);new_line;

            end if;
        end loop;

    end loop;

    put(count);put(" primes"); new_line;

end zprime;

--pragma suppress_all;
```

```
      PROGRAM ZPRIME

C   zprime.for   Eratosthenes Sieve Prime Number Program in VAX FORTRAN.

      logical flags( 8191 )
      integer i, iprime, k, icnt, iter



      write( 6,900 )
  900 format(’ 10 iterations’)

      do 92 iter = 1,10
      icnt = 0

      do 10 i = 1,8191
   10 flags( i ) = .true.

      do 91 i = 1,8191
      if ( flags(i) .eq. .false. ) go to 91

      iprime = i + i + 1
      k = i + iprime

   20 if( k .gt. 8191 ) go to 90

      flags( k ) = .false.
      k = k + iprime
      go to 20

   90 icnt = icnt + 1
c     type *,icnt,iprime

   91 continue

   92 continue

      write( 6,901 ) icnt
  901 format( 1x,i6,’ primes’)

      end
```

```
/* zprime.c  Eratosthenes Sieve Prime Number Program in C for MicroVAX II. */

#define    TRUE  1
#define    FALSE 0
#define    SIZE  8191

char flags[SIZE];

main()
{
    int i,prime,k,count,iter;

    printf("10 iterations\n");

    for( iter =1;iter <= 10; iter++)
    {
        count=0;
        for(i = 0;i < SIZE; i++)
            flags[i] = TRUE;

        for(i = 0;i < SIZE; i++)
        {
            if(flags[i])
            {
                prime = i + i + 3;
                k = i + prime;
                while( k < SIZE )
                {
                    flags[k] = FALSE;
                    k += prime;
                }
                ++count;
            }
        }
    }
    printf("%d primes\n",count);
}
```

```
;  zprime.mar  -  Eratosthenes Sieve Prime Number Program in VAX Macro.

        .TITLE  ZPRIME

        .psect  data,noexe
FLAGS:  .BLKW   8192
DUMMY:  .BLKW   1       ; THIS IS USED WITH THE MOVC5 INSTRUCTION

ttchan: .blkw   1
ttdesc: .long   20$-10$
        .long   10$
10$:    .ascii  /tt:/
20$:
iostat: .blkq   1

strmsg: .ascii  "10 iterations "<13><10>
stpmsg: .ascii  "_____ primes"<13><10>
prmmsg: .ascii  "prime= _____"<13><10>

quot:   .long   0
rem:    .long   0


        .psect  code,nowrt,exe
        .ENTRY  ZPRIME,^M<R2,R3,R4,R5,R6,R7,R8,R9,R11>  ; set entry mask

        $assign_s devnam=ttdesc,-
                chan=ttchan

;** write out the number of iterations (in R2) here

        $qiow_s chan=ttchan,-
                func=#io$_writevblk,-
                iosb=iostat,-
                p1=strmsg,-
                p2=#16

        MOVW    #1,R3           ; initialize iteration counter

        MOVZBL  #10,R2          ; set number of iterations

        MOVZWL  #8191,R4        ; set size of FLAGS array

        CLRL    R7              ; clear "k" index for FLAGS array

; do for R2 iterations
L1:
        CLRW    R11             ; clear count of number of primes found

        pushr   #^m<r2,r3,r4>   ; Save registers that MOVC5 uses
                                ; set FLAGS array to all true (non-zero)
        MOVC5   #0,DUMMY,#^XFF,#16384,FLAGS
        popr    #^m<r2,r3,r4>   ; restore registers


        MOVL    #1,R5           ; reset array index and loop counter
L3:
        TSTW    FLAGS[R5]       ; test FLAGS for prime (true for first time)
        BEQL    S91

        ADDW3   R5,R5,R6        ; calculate the value of the prime from
        INCW    R6              ; its index, put in R6

        ADDW3   R6,R5,R7        ; calculate the prime's first odd multiple

S20:
        CMPW    R7,#8191
        BGTR    S90

        CLRW    FLAGS[R7]       ; set the odd multiple’s flag value false
        ADDW2   R6,R7           ; calculate next odd multiple of the prime
        BRB     S20

S90:
        INCW    R11             ; increment count of primes found

;**             write out the value of the prime (in R6) here (for debug)
;
;       cvtwl   r6,r0
;       movab   prmmsg+7,r8
;       bsbw    convert
;
;       $qiow_s chan=ttchan,-
;               func=#io$_writevblk,-
;               iosb=iostat,-
;               p1=prmmsg,-
;               p2=#14

S91:
        ACBW    R4,#1,R5,L3     ; all through FLAGS array?

        ACBW    R2,#1,R3,L1     ; all through with number of iterations?

;**             write out the number of primes found (in R11) here

        cvtwl   r11,r0
        movab   stpmsg,r8
        bsbw    convert

        $qiow_s chan=ttchan,-
                func=#io$_writevblk,-
                iosb=iostat,-
                p1=stpmsg,-
                p2=#14

        MOVL    #1,R0           ; set successful completion (VAX requires)
        RET
```

```
;
; Convert binary integer to ascii string, 5 digits, right-justified, zero-
; filled
;                       assume binary value is in work register r0
;                       and address of ascii string to receive converted
;                       integer is in register r8

convert:
;  msg+0 byte
;       clrl    rl              ;this instruction needs to be here,
                                ;(it can't go after the ediv
                                ; instruction either).
        ediv    #10000,r0,quot,rem
        addl2   #48,quot
        cvtlb   quot,r0
        movb    r0,(r8)+

;  msg+1 byte
        movl    rem,r0
        ediv    #1000,r0,quot,rem
        addl2   #48,quot
        cvtlb   quot,r0
        movb    r0,(r8)+

;  msg+2 byte
        movl    rem,r0
        ediv    #100,r0,quot,rem
        addl2   #48,quot
        cvtlb   quot,r0
        movb    r0,(r8)+

;  msg+3 byte
        movl    rem,r0
        ediv    #10,r0,quot,rem
        addl2   #48,quot
        cvtlb   quot,r0
        movb    r0,(r8)+

;  msg+4 byte
        addl2   #48,rem
        cvtlb   rem,r0
        movb    r0,(r8)+

        rsb

        .END    ZPRIME
```

```
(*  zprime.p  *)

(* Eratosthenes Sieve Prime Number Program in PASCAL for the SUN. *)

program zprime( output );

const
    size = 8190;

var
    flags : array[ 0..size ] of boolean;
    i, prime, k, count, iter : integer;

begin          (* ZPRIME *)

    writeln( output, '10 iterations' );

    for iter := 1 to 10 do
    begin

        count := 0;
        for i := 0 to size do
            flags[ i ] := true;

        for i := 0 to size do
        if flags[ i ]  then
        begin

            prime := i + i + 3;
            k := i + prime;

            while k <= size do
            begin
                flags[ k ] := false;
                k := k + prime;
            end;

            count := count + 1;        (* Count of number of prime numbers *)

        end;
    end;

    writeln( output,count,' primes' );

end.           (* ZPRIME *)
```

```
|  zprime.s

| Erathosthenes sieve prime number program in Motorola 68000 assembly.
| This is the version for the SUN.


                .data

                .comm       flags,16382  | Reserve 8191 words.
rem:            .word       0
quotient:       .word       0

                                         | Build output messages.

msg1:           .asciz      "10 iterations\12"

msg2:           .asciz      "     primes\12"

                .text

                pea         msg1         | Print "10 iterations".
                jsr         _printf      | Jump to the print routine.

                lea         flags,a2     | Load beginning address of flags.
                clrl        d6           | Clear register d6.
                movw        #10,d5       | Upper loop counter for 1-10 loop.
                clrl        d1           | Lower loop counter for 1-10 loop.
                movw        #8191,d2     | Upper loop counter for array flags.

l1:             addw        #1,d1        | Increment lower loop ctr for 1-10 loop
                clrl        d7           | Count.
                clrl        d3           | Lower loop ctr. for array flags.

| Set all 8191 cells of array flags to true ( i.e. = 1 ).

                lea         flags,a1     | Load beginning addr. of flags.

l2:             addw        #1,d3        | Increment lower loop counter.
                movw        #1,a1@+      | Set to true. Point to next cell.
                cmpw        d3,d2        | See if the loop is complete.
                bgt         l2           | Loop until d2=d3 (8191).

| Array flags has been set to true. Continue.

                clrl        d3           | Clear register d3. ( set to  zero ).
                lea         flags,a1     | load beginning address of flags.

| Begin 1-8191 loop. Check each cell to see if = true (=1).

l3:             addw        #1,d3        | Increment lower loop counter.
                cmpw        #1,a1@+      | See if this cell = true.
                blt         l5           | Not true, check the next cell.

| This cell of flags array is true.

                movw        d3,d6        | d3 contains i.
                addw        d3,d6        | d6 now contains i + i.
                addw        #1,d6        | d6 now contains i + i + 1.

| d6 contains PRIME and d4 contains K.

                movw        d3,d4        | k = i + prime.
                addw        d6,d4

| Loop while k lt or = 8191.
```

```
14:             cmpw      d4,d2          | d2 contains the value 8191.
                blt       17             | k is now gt 8191.

                movw      d4,d0
                aslw      #1,d0          | Shift contents of d0 left 1 bit.
                subw      #2,d0          | Subtract 2 from contents of d0.

                                         | Set this cell to false.

                movw      #0x00,a2@(0,d0:w)
                addw      d6,d4          | k = k + prime.
                bra       14             | Check the next location.

| K has become gt 8191.  Increment count.

17:             addw      #1,d7          | Increment prime number counter.

15:             cmpw      d3,d2          | Compare lower loop ctr. to 8191.
                bgt       13             | Continue loop. See if next cell true.

| Have looped 8191 times. See if outer loop has looped 10 times.

16:             cmpw      d1,d5          | Compare lower loop counter to 10.
                bgt       11             | Execute next iteration of 1-10 loop.

| All 10 iterations of the outer loop have been completed, finished.
| Output the number of primes ( count ) that were found.

                lea       msg2,a4        | Load addr. of the return buffer.
                jmp       convert        | Convert to the ASCII equivalent.

back:           pea       msg2           | Print out the number of primes.
                jsr       _printf        | Jump to the print routine.

return:         jsr       _exit          | Exit the program.

|               This routine converts a binary value to its ASCII equivalent.
|               This routine assumes a maximum of five (5) characters.

|               On entry to this routine, register d7 contains the binary
|               value to be converted and register a4 contains the address
|               of the buffer where the ASCII equivalent is to be placed.

convert:        divs      #10000,d7
                movl      d7,rem         | Save the remainder from the division.
                cmpw      #0x00,d7       | See if this digit is zero.
                ble       next           | If so, skip to the next digit.
                addw      #060,quotient| Add an ASCII zero to the answer.
                movb      quotient+1,a4@+

next:           clrl      d7
                movw      rem,d7
                divs      #1000,d7
                movl      d7,rem
                addw      #060,quotient
                movb      quotient+1,a4@+

                clrl      d7
                movw      rem,d7
                divs      #100,d7
                movl      d7,rem
                addw      #060,quotient
                movb      quotient+1,a4@+

                clrl      d7
                movw      rem,d7
```

```
divs        #10,d7
movl        d7,rem
addw        #060,quotient
movb        quotient+1,a4@+

addw        #060,rem
movb        rem+1,a4@+

jmp         back            | Return to the caller.
```

```
-- fpatestsub.ada   floating point test benchmark (w/subroutine), MicroVAX Ada.

with text_io;   use text_io;

procedure fpatestsub is

    package realnum is new float_io( float );
    use realnum;

    x   : float := 100.0;
    y   : float := 100.0;
    z   : float := 100.0;

    pragma optimize( time );

--***************************
    procedure fpa(x: in out float;
                  y: in out float;
                  z: in out float) is
    begin
        y := x / x;
        z := (y/x) + 1.0;
    end fpa;
--***************************

begin

    put("start"); new_line;

    for i in 1..100000  loop
        fpa(x,y,z);
    end loop;

    put("z = ");  put( z,4,30,0 );  new_line;
end fpatestsub;

pragma suppress_all;
```

```
c  fpatestsub.for   -- floating point test benchmark in FORTRAN, MicroVAX  II.

        program fpatest

        real*4  x,y,z

        data x /100./ y /100./ z /100./

        type *,'start'

        do  i=1,100000
            call fpa(x,y,z)
        enddo

        type 900, z
 900    format(' z = ',f34.30)

        end


        subroutine fpa(x,y,z)
        real*4 x,y,z
        y=x/x
        z=y/x + 1.
        return
        end
```

```
/* fpatestsub.c  --  floating point test benchmark in "C", MicroVAX II. */

main()
{
    float x = 100.;
    float y = 100.;
    float z = 100.;

    int i;

    printf("start\n");

    for( i = 0; i < 100000; i++ )
        fpa( &x, &y, &z);

    printf(" z = %34.30f\n",z );

}


fpa(x,y,z)
float *x, *y, *z;
{
    *y = *x / *x;
    *z = ( *y / *x ) + 1.0;
}
```

```
--   speed_test.ada  -  floating point test benchmark ( w/sine ),
--                      VAX Ada.

with text_io;  use text_io;
with float_math_lib;  use float_math_lib;

procedure speed_test is

    package debug_io is new float_io( float );
    use debug_io;

    sum : float := 0.0;
    x   : float := 0.0;

begin

    put("10000 iterations");  new_line;

    for i in 1..10000  loop
        x := float( i );
        sum := sum + 1.0 / ( x + sin(x) );
    end loop;

    put("done");  new_line;

    put(" sum = ");  put( sum,2,6,0 );  new_line;        -- for debug

end speed_test;


--    pragma suppress_all;
--    speed_testsav.ada

--    speed_test.ada  -  floating point test benchmark ( w/sine ), VAX
--                       Ada; special version for long_float d_float.


-- The VAX/VMS predefined instantiation of math_lib for long_float
-- is not compatiable with the d_float pragma and requires user
-- instantiation of math_lib for long_float types.

pragma long_float( d_float );

with text_io;  use text_io;

with math_lib;

procedure speed_testsav is

                                -- user instantation of math_lib.

    package my_math_lib is new math_lib( long_float );
    use my_math_lib;

    package debug_io is new float_io( long_float );
    use debug_io;

    sum : long_float := 0.0;
    x   : long_float := 0.0;

begin

    put("10000 iterations");  new_line;

    for i in 1..10000  loop
        x := long_float( i );
        sum := sum + 1.0 / ( x + sin(x) );
    end loop;

    put("done");  new_line;

    put(" sum = ");  put( sum,2,6,0 );  new_line;      -- for debug

end speed_testsav;


--    pragma suppress_all;
```

```
c  speed_test.for  -  floating point test benchmark ( w/sine ), MicroVAX II.

      program speed_test

      real*4 x, sum


      write(5,700)
  700 format(' 10000 iterations')

      do 10 i = 1,10000
          x = i
   10     sum = sum + 1.0 / ( x + sin(x) )

      write(6,800)
  800 format(' done')
c                              for debug
      write(5,900)sum
  900 format(' sum=',f9.6)

      end
```

```
/* speed_test.c  -  floating point test benchmark ( w/sine ), MicroVAX II. */

#include math

main()
{
    int i;
    float sum= 0., x = 0.;

    printf(" 10000 iterations\n");

    for( i = 1; i <= 10000; i++ )
    {
        x = i;
        sum = sum + 1. / ( x + sin(x) );
    }

    printf("done\n");

    printf("sum = %f\n",sum);

}
```

```
-- run5.ada  -  Matrix manipulation test benchmark, MicroVAX II.
--              This version uses separate compilation of subprograms
--              ludecm and prnary.

with text_io;  use text_io;

procedure run5 is

    type buf is array( integer range <>,integer range <> ) of float;

    n  : integer := 4;
    n2 : integer := 2;

    a  : buf(1..n,1..n)   := (( -2.0,-4.0,-6.0,-8.0 ),
                              ( 2.0,5.0,8.0,11.0 ),
                              ( 1.0,5.0,10.0,15.0 ),
                              ( 5.0,6.0,5.0,5.0 ));
    a1 : buf(1..n,1..n );
    b  : buf(1..n2,1..n2) := (( 1.0,1.0 ),
                              ( 2.0,3.0 ));
    b1 : buf(1..n2,1..n2 );

    procedure ludecm( array1 : buf;
                      number : integer;
                      array2 : in out buf ) is separate;

    procedure prnary( array2 : in out buf;
                      num : integer ) is separate;

begin                                       -- beginning of run5.

    put(" 10,000 iterations");  new_line;

    for i in 1..10000  loop
        ludecm( a,n,a1 );
        ludecm( b,n2,b1 );
    end loop;

    put(" done!");  new_line;


    prnary( a,n );
    prnary( b,n2 );
    prnary( a1,n );
    prnary( b1,n2 );

end run5;                                   -- end of run5.

--pragma suppress_all;
```

```
-- ludecm.ada

separate ( run5 )

procedure ludecm( array1 : buf; number : integer; array2 : in out buf ) is

--  Lower and upper decomposition of square matrix
--  with Gaussian elimination.
--  Tests floating point computation, array addressing,
--  and nested iteration constructs.

    mult   : float;
    diag   : integer;
    row    : integer;
    col    : integer;

begin                                   -- beginning of ludecm.

    array2 := array1;

    for diag in 1..number-1  loop
        for row in diag+1..number  loop
            mult := array2( diag,row ) / array2( diag,diag );
            array2( diag,row ) := mult;
            for col in diag+1..number  loop
                array2(col,row) := array2(col,row) - mult*array2(col,diag);
            end loop;
        end loop;
    end loop;

end ludecm;                             -- end of ludecm.

--pragma suppress_all;
```

```
-- prnary.ada

with text_io;  use text_io;

separate( run5 )

procedure prnary( array2 : in out buf; num : integer ) is

    package buffer_io is new float_io( float );
    use buffer_io;

    package new_integer_io is new integer_io( integer );
    use new_integer_io;

begin

    new_line;  new_line;

    for i in 1..num  loop
        for j in 1..num  loop
            put("array(");  put(i,1);  put(",");
            put(j,1);  put(")=");  put( array2(i,j),3,0,0 );
            new_line;
        end loop;
    end loop;

    new_line;

end prnary;

--pragma suppress_all;
```

```
c  run5.for  - Matrix manipulation test benchmark, MicroVAX II.

        program run5

        real*4  a(4,4),b(2,2)
        real*4  a1(4,4),b1(2,2)

        data a/-2.,-4.,-6.,-8.,
        1       2., 5., 8.,11.,
        2       1., 5.,10.,15.,
        3       5., 6., 5., 5./

        data b/1.,1.,
        1      2.,3./

        write(6,900)
 900    format(' 10,000 iterations')

        do 10 i=1,10000

            call ludecm(a,4,a1)
            call ludecm(b,2,b1)

 10     continue

        write(6,901)
 901    format(' done!')

        call prnary(a,4)
        call prnary(a1,4)
        call prnary(b,2)
        call prnary(b1,2)

        end
```

```
      subroutine ludecm(array1,n,array2)

c  Lower and upper decomposition of square matrix
c  with Gaussian elimination.
c  Tests floating point computation, array addressing,
c  and nested iteration constructs.

      real*4 array1(n,n),mult,array2(n,n)
      integer*2 diag,row,col


c                       copy input array to the working array
      do 9 row= 1,n
          do 10 col=1,n
              array2(col,row)=array1(col,row)
 10       continue
 9    continue


      do 39 diag= 1,n-1

          do 29 row= diag+1,n

              mult=array2(row,diag)/array2(diag,diag)
              array2(row,diag)=mult

              do 19 col= diag+1,n

                  array2(row,col)=array2(row,col)-mult*array2(diag,col)

 19           continue

 29       continue

 39   continue


      return

      end
```

```
        subroutine prnary(array,n)

c  subroutine to print out the array.

        real*4 array(n,n)
        integer*2  row,col


        do 20 row= 1,n

            do 10 col=1,n

                write(6,900)col,row,array(col,row)
 900            format(' array(',i1,',',i1,')=',f4.0)

 10         continue

 20     continue


        write(6,901)
 901    format(' ')

        return
        end
```

```
/* run5.c  - Matrix manipulation test benchmark, MicroVAX II. */

float a[4][4]=
{
                {-2.,-4.,-6.,-8.},
                { 2., 5., 8.,11.},
                { 1., 5.,10.,15.},
                { 5., 6., 5., 5.},
};
float b[2][2]=
{
                { 1., 1.},
                { 2., 3.},
};
float a1[4][4],b1[2][2];

main()
{
    int iter;

    printf("10000 iterations\n");

    for ( iter=1; iter<=10000; ++iter )
    {
        ludecm( a,4,a1 );
        ludecm( b,2,b1 );
    }

    printf("done!\n");

    prnary(a,4);
    prnary(b,2);
    prnary(a1,4);
    prnary(b1,2);
}

ludecm(array,n,array1)

/*  Lower and upper decomposition of square matrix      */
/*  with Gaussian elimination                           */
/*  tests floating point computation, array addressing, */
/*  and nested iteration constructs.                    */

int n;

/*  Treat arrays as single dimensioned because "C"  */
/*  does not support variable 2-dimensioned arrays. */

float array[],array1[];
{
    int diag,row,col,i;
    float mult;

    for( i=0; i<=((n-1)*n+(n-1)); i++)
        array1[i] = array[i];

    for(diag=0; diag<(n-1); ++diag )
    {
        for( row=diag+1; row<n; ++row )
        {

/*                      Use algorithm to compute array element   */
/*                      to simulate 2-dimensional array          */

            mult = array1[diag*n+row]/array1[diag*n+diag];
            array1[diag*n+row]=mult;
            for( col=diag+1; col<n; ++col )
                array1[col*n+row]=array1[col*n+row]-mult*array1[col*n+diag];
        }
    }
}

prnary(array,n)

/*  Routine to print out the array. */

int n;
float array[];

{
    int i,j;

    for( j=0; j<n; ++j )
    {
        for( i=0; i<n; ++i )
            printf("array[%d][%d]= %5.1f\n",j,i,array[j*n+i]);
    }
    printf("\n");
}
```

```
c  tstrun5.for  -  Benchmark program to test multi-dimensioned
c                  arrays, in FORTRAN for the MicroVAX II.

        program tstrun5

        real*4  a(4,4),b(2,2)
        real*4  a1(4,4),b1(2,2)

        data a/-2.,-4.,-6.,-8.,
        1       2., 5., 8.,11.,
        2       1., 5.,10.,15.,
        3       5., 6., 5., 5./

        data b/1.,1.,
        1      2.,3./

        write(6,900)
 900    format(' 10,000 iterations')

        do 10 i=1,10000

            call ludecm(a,4,a1)
            call ludecm(b,2,b1)

 10     continue

        write(6,901)
 901    format(' done!')

        call prnary(a,4)
        call prnary(a1,4)
        call prnary(b,2)
        call prnary(b1,2)

        end
        subroutine ludecm(array1,n,array2)

c  Lower and upper decomposition of square matrix
c  with Gaussian elimination.
c  Tests floating point computation, array addressing,
c  and nested iteration constructs.

        real*4 array1(1),mult,array2(1)
        integer*2 diag,row,col


c                       copy input array to the working array
        do 9 i= 1,n*n
                array2( i )=array1( i )
 9      continue

       do 39 diag= 1,n-1

        do 29 row= diag+1,n
        mult=array2(row+n*(diag-1))/array2(diag+n*(diag-1))
        array2(row+n*(diag-1))=mult

            do 19 col= diag+1,n
            array2(row+n*(col-1))=array2(row+n*(col-1)) -
        1                            mult*array2(diag+n*(col-1))
 19         continue

 29     continue

 39    continue


        return

        end
        subroutine prnary(array,n)

c       subroutine to print out the array.

        real*4 array(n,n)
        integer*2  row,col


        do 20 row= 1,n

            do 10 col=1,n

                write(6,900)col,row,array(col,row)
 900            format(' array(',i1,',',i1,')=',f4.0)

 10         continue

 20     continue


        write(6,901)
 901    format(' ')

        return
        end
```

```
-- timtes.ada  -  Disk write timing benchmark, MicroVAX II.

with text_io;        use text_io;
with calendar;       use calendar;
with sequential_io;

procedure timtes is

    type buffer is array( short_integer range 1..2048 ) of short_integer;

    package buffer_io is new sequential_io( buffer );
    use buffer_io;

    package duration_text_io is new fixed_io( duration );
    use duration_text_io;

    data_file  : buffer_io.file_type;
    ibuf       : buffer;
    i          : short_integer;
    t1         : duration;
    t2         : duration;
    deltime    : duration;
    date       : time;

    pragma optimize( time );

begin                                  -- start of the Ada program timtes.

    new_line;
    put( " Program TIMTES - MicroVAX ADA version ");
    new_line;

    for i in short_integer range 1..2048  loop
        ibuf( i ) := i;
    end loop;

    create( file => data_file,         -- create a file.
            name => "timetest.dat",
                                       -- the "form" parameter is
                                       -- implementation dependent.
            form => "file;"             &
            "best_try_contiguous yes;"  &
            "allocation 2500;"          );

    date := clock;                     -- get current value of time.
    t1 := seconds( date );             -- get begin time (t1) in seconds.

    for i in 1..300  loop
        write( data_file, ibuf );      -- write 300 records.
    end loop;

    date := clock;                     -- get current value of time.
    t2 := seconds( date );             -- get end time (t2) in seconds.

    deltime :=  t2 - t1;               -- calculate the delta time in seconds.

    close( data_file );                -- close the file.

    put  ( " time difference = " );
    put  ( deltime );                  -- print out the delta time.
    new_line;


end timtes;
```

```
c   timtes.for  -  Disk write timing benchmark in FORTRAN, MicroVAX II.

      program timtes

      integer*2 ibuf( 2048 )
      integer i
      real t1, delta


c                                   output the program header to crt.

      write( 6,100 )
  100 format( /,' Program TIMTES - MicroVAX II FORTRAN version ',/ )

c                                   load the buffer "ibuf".

      do 10 i = 1,2048
      ibuf( i ) = i
   10 continue

c                                   open the disk file "timetest".

      open( unit=4,file='timetest.dat',
     1        status='unknown',form='unformatted',access='direct',
     1        organization='sequential',
     1        initialsize=2500,recordsize=1024 )

      t1 = secnds( 0.0 )            ! get the start time.

c                                   write 300 records to "timetest".

      do 20 i = 1,300
      write( 4, rec=i ) ibuf
   20 continue

      delta = secnds( t1 )          ! get the delta time (in seconds).

c                                   output the delta time to the crt.

      write( 6,200 ) delta
  200 format( 1x,' time difference = ', f10.3,/ )

      close( unit=4 )               ! close the disk file "timetest".

      call exit                     ! exit the program.
      end
```

```
/*  timtes.c -  Disk write timing benchmark, MicroVAX II */


#define NREC 300                        /* Number of records to write.  */
#define N    4096                       /* Number of bytes per record.  */
#define NW   2048                       /* Number of words per record.  */
#define PMODE 0777                      /* Mode with which to open file */

#include perror

short int ibuf[ NW ], fd;
long t1, t2, delta, time();

main()
{
    int i, nwritten;

    for( i = 0; i < NW; ++i )           /* Initialize the write buffer. */
        ibuf[ i ] = i + 1;

    fd = creat("timetest.dat", PMODE,"mrs=4096","rfm=fix","alq=2400","fop=ctg");

    if( fd != -1 )
    {
        t1 = time((long *) 0 );         /* get starting time, seconds.  */

        for( i = 1; i <= NREC; ++i )    /* write NREC records.          */
            nwritten = write( fd, ibuf, sizeof(ibuf) );

        t2 = time((long *) 0 );         /* Get ending time, seconds.    */
        delta = t2 -t1;                 /* Calculate the delta time.    */
        printf(" nwritten= %d time diff= %ld \n",nwritten,delta);
    }
    else
    {
        printf(" error creating the file \n");
        perror("timtes");
    }
}
```

```
(*  timtes.p  --  Disk write benchmark for SUN Pascal  *)

program timtes ( output,testdat );

(* A PASCAL program to test writing to a disk file *)

const
    nrec = 300;                         (* Number of records to write *)
    nw   = 1024;                        (* Number of (32 bit) words per record *)

type
    ibuf = record
           data : array[1..nw] of integer;
           end;

var
    newdata : ibuf;
    testdat : file of ibuf;             (* File for the test data *)
    i       : integer;                  (* Loop counter *)
    btime   : integer;                  (* Begin time *)
    etime   : integer;                  (* End time *)
    delta   : real;                     (* Delta time in seconds *)

    procedure sync;                     external c;


begin         (* TIMTES *)

                                        (* Create the file for the test data *)

    rewrite( testdat,'testdat.tf' );

    for i := 1 to nw  do
        newdata.data[ i ] := i;

    btime := wallclock;                 (* Get the current system time *)

    testdat^ := newdata;

    for i := 1 to nrec  do
    begin
        put( testdat );                 (* Put the test data in the file *)


                                        (* Write output buffered for the PASCAL
                                           file testdat into the UNIX file *)

(*        flush( testdat );     *)

        sync;                           (* Call the "C" function "sync" *)
    end;

    etime := wallclock;                 (* Get the current system time *)
    delta := etime - btime;             (* Calculate delta time in seconds *)

                                        (* Output the time to the screen *)

    writeln( output,' Time for write = ',delta:7:3,' seconds');

end.          (* TIMTES *)
```

```
c  timtesblok.for
      program timtesblok

c          Fortran program to time writing to the disk.

      integer*2 ibuf( 2048 )
      integer*4 i
      real*4 t1, delta

      integer*4  rabadr,for$rab,rmssts,rmsstv,sys$write,sys$put

      integer*4  uopen
      external   uopen

c                                      output the program header to crt.

      write( 6,100 )
  100 format( /,' Program TIMTESFOR - MicroVAX II FORTRAN version ',/ )

c                                      load the buffer "ibuf".

      do 10 i = 1,2048
      ibuf( i ) = i
   10 continue

c                                      open the disk file "testfile".

      open( unit=4,file='timetest.dat',
        1       status='new',form='unformatted',
        1       organization='sequential',
        1       recordtype='fixed',
        1       initialsize=2500,recl=1024,useropen=uopen,
        1       access='sequential',
        1       err=99 )

      t1 = secnds( 0.0 )               ! get the start time.
c                                      write 300 records to "testfile".

      rabadr = for$rab(4)

      call setrab(%val(rabadr),ibuf)

      do 20 i = 1,300
      ids = sys$write(%val(rabadr))
   20 continue


      delta = secnds( t1 )             ! get the delta time (in seconds).
c                                      output the delta time to the crt.

      write( 6,200 ) delta
  200 format( 1x,' time difference = ', f10.3,/ )

        close(unit=4)

      call exit                        ! exit the program.
 99   call errsns(,rmssts,rmsstv,,)
      call sysmsg(rmssts)
      call sysmsg(rmsstv)
      end
```

```
c  uopen.for  -  user-open routine.  Called in 'open' statement

        integer*4 function uopen(fab,rab,lun)

        include '($fabdef)'
        include '($rabdef)'

        record /fabdef/fab
        record /rabdef/rab

        integer*4  lun,chan,sys$create,sys$connect


        fab.fab$b_fac = fab.fab$b_fac .or. fab$m_bio    !set user-open bit

        uopen = sys$create(fab)                         !open the file
        if( .not. uopen ) return

        uopen = sys$connect(rab)
        if( .not. uopen ) return

        return
        end


        subroutine setrab(rab,ibuf)

        include '($rabdef)'

        record /rabdef/rab

        integer*2 ibuf(1)

        rab.rab$w_rsz = 4096
        rab.rab$l_rbf = %loc(ibuf)


        return
        end
```

```
c  timtesblokput.for  -  Fortran program to time writing to the disk.

      program timtesblok

      integer*2 ibuf( 2048 )
      integer*4 i
      real*4 t1, delta

      integer*4  rabadr,for$rab,rmssts,rmsstv,sys$put

      integer*4  uopen
      external   uopen

c                                      output the program header to crt.

      write( 6,100 )
  100 format( /,' Program TIMTESFOR - MicroVAX II FORTRAN version ',/ )

c                                      load the buffer "ibuf".

      do 10 i = 1,2048
      ibuf( i ) = i
   10 continue

c                                      open the disk file "testfile".

      open( unit=4,file='timetest.dat',
        1       status='new',form='unformatted',
        1       organization='sequential',
        1       recordtype='fixed',buffercount=22,
        1       initialsize=2500,recl=1024,useropen=uopen,
        1       access='sequential',
        1       err=99 )

      t1 = secnds( 0.0 )               ! get the start time.
c                                      write 300 records to "testfile".

      rabadr = for$rab(4)

      call setrab(%val(rabadr),ibuf)

      do 20 i = 1,300
      ids = sys$put(%val(rabadr))
   20 continue


      delta = secnds( t1 )             ! get the delta time (in seconds).
c                                      output the delta time to the crt.

      write( 6,200 ) delta
  200 format( 1x,' time difference = ', f10.3,/ )

        type *,'records written: ',i,' status:'
        call sysmsg(ids)

        close(unit=4)

      call exit                        ! exit the program.
 99   call errsns(,rmssts,rmsstv,,)
      call sysmsg(rmssts)
      call sysmsg(rmsstv)
      end
```

```
c  uopen.for  -  user-open routine.  Called in 'open' statement

        integer*4 function uopen(fab,rab,lun)

        include '($fabdef)'
        include '($rabdef)'

        record /fabdef/fab
        record /rabdef/rab

        integer*4  lun,chan,sys$create,sys$connect


        fab.fab$b_fac = fab.fab$b_fac                   !set user-open bits

        uopen = sys$create(fab)                         !open the file
        if( .not. uopen ) return

        uopen = sys$connect(rab)
        if( .not. uopen ) return

        return
        end


        subroutine setrab(rab,ibuf)

        include '($rabdef)'

        record /rabdef/rab

        integer*2 ibuf(1)

        rab.rab$w_rsz = 4096
        rab.rab$l_rbf = %loc(ibuf)


        return
        end
```

```
-- timetask1.ada  -  Rendezvous Response time benchmark, MicroVAX-II.

--                This is a two-task response time test for the purpose
--                of determining the overhead required by the system for
--                a rendezvous. A rendezvous is used to enable the
--                synchronization of the two tasks in order to give an
--                idea of how fast a task can respond to being started
--                by another task.

with text_io;  use text_io;

procedure timetask1 is

    i          : integer;

    task timetask2 is
        entry start;
    end timetask2;

    task body timetask2 is

    begin

        loop

            select
                accept start;
            or
                terminate;
            end select;

        end loop;

    end timetask2;

begin

    put("start");  new_line;

    for i in 1..1000 loop
        timetask2.start;                        -- start timetask2.
    end loop;

    put("stop");  new_line;

end timetask1;
```

```
-- adasend.ada  -  Two-task synchronized data tansfer (send/receive)
--                 benchmark, MicroVAX-II.

with text_io;  use text_io;

procedure adasend is

    type buffer is array( short_integer range 1..2048 ) of short_integer;

    package debug_io is new integer_io( short_integer );
    use debug_io;

    i          : short_integer;
    k          : short_integer;
    j          : short_integer := 0;
    l          : short_integer;
    ibuf       : buffer;

--*******************************************************************

    task adarecv is
        entry start;
    end adarecv;


    task body adarecv is

    begin                             -- beginning of adarecv.

        loop                          -- loop forever.

            j := j + 1;               -- increment loop counter.

            select

                accept start do

                    for l in short_integer range 1..2048  loop

                        if ibuf( l ) /= j  then
                            put(" data is incorrect for loop = " );
                            put( j );  new_line;
                        end if;

                    end loop;

                end start;

            or

                terminate;            -- terminate adarecv.

            end select;

        end loop;

    end adarecv;                      -- end of the receive task.

--*******************************************************************

begin                                 -- beginning of adasend.

    put(" The send task (Program ADASEND) is starting");  new_line;  new_line;
```

```
     for k in short_integer range 1..300  loop

         for i in short_integer range 1..2048  loop
             ibuf( i ) := k;
         end loop;

         adarecv.start;

     end loop;

     put(" Program ADARECV is exiting");

end adasend;
```

```
-- adasend_disk.ada  - Two-task data transfer (send/receive)
--                     via shared disk file, MicroVAX-II.

with text_io;  use text_io;
with direct_io;

procedure adasend_disk is

    type buffer is array( short_integer range 1..2048 ) of short_integer;

    package buffer_io is new direct_io( buffer );
    use buffer_io;

    package debug_io is new integer_io( short_integer );
    use debug_io;

    data_file  : buffer_io.file_type;
    ibuf       : buffer;
    i          : short_integer;
    k          : buffer_io.positive_count;
    j          : short_integer := 0;
    l          : short_integer;
    recno      : buffer_io.positive_count;
    numrec     : short_integer := 0;

    pragma volatile( numrec);
    pragma volatile( j );

--*********************************************************************

    task adarecv_disk;

    task body adarecv_disk is

    ibuf       : buffer;

    begin                                   -- beginning of adarecv.

        loop                                -- loop forever.

        j := j + 1;                         -- increment loop counter.

            while numrec < j  loop
                delay 0.01;
            end loop;

            read( data_file,ibuf,buffer_io.positive_count( j ) );

                for l in short_integer range 1..2048  loop

                    if ibuf( l ) /= j  then
                        put(" data is incorrect for loop = " );
                        put( j );  new_line;
                    end if;

                end loop;

        end loop;

        exception
            when buffer_io.end_error =>
            put("adarecv_disk: end_error");  new_line( 2 );

            when buffer_io.use_error =>
            put("adarecv_disk: use_error");  new_line( 2 );
            when buffer_io.data_error =>
            put("adarecv_disk: data_error");  new_line( 2 );

            when others =>
            put("adarecv_disk: unknown error");  new_line( 2 );

    end adarecv_disk;                  -- end of the receive task.

--********************************************************************

begin                                  -- beginning of adasend.

    create( file => data_file,
            name => "dua2:[user.horne]test.dat",
            form => "file;"                                &
            "best_try_contiguous yes;"                     &
            "allocation 2500;"                             );

    new_line;
    put("ADASEND_DISK is starting");  new_line( 2 );

    for k in buffer_io.positive_count range 1..300  loop

        recno := k;

        for i in short_integer range 1..2048  loop
            ibuf( i ) := short_integer ( k );
        end loop;

        write( data_file,ibuf,recno );
        numrec := short_integer( k );

    end loop;

    while j <= numrec  loop
        delay 0.01;
        if adarecv_disk'terminated  then
            new_line( 2 );
            put("ADARECV_DISK task terminated abnormally");
            new_line;
            put("read error on record number ");  put( j );  new_line;
            exit;
        end if;
    end loop;

    if j > numrec  then
        put("ADARECV_DISK task terminated normally");  new_line;
        put( numrec );  put(" records were read");  new_line( 2 );
        abort adarecv_disk;
    end if;

end adasend_disk;
```

```
c  forsend.for  -  Send task for 2-task synchronized data transfer.
c               --  event flag/installed shareable common version

        program forsend

        integer*2  ibuf,iflag
        integer*2  i,k

        integer*4  ispawn,ids,sys$ascefc,sys$setef,sys$waitfr,sys$clref

        external cli$m_nowait

        common/comglb/ibuf(2048),iflag

        ids = sys$ascefc(%val(64),'efcluster',,)        ! assign ef cluster


c                       start up the receive task to run concurrently
        ispawn=%loc(cli$m_nowait)
        call lib$spawn('run forrecv',,,ispawn,'forrecv',ipid)

c                       let receive task start
        call wait('0 ::3.0',ids)


c                       -- beginning of forsend

        type *,' The send task (Program FORSEND) is starting'

        do 30 k=1,300

c                       fill buffer
            do 20 i=1,2048
                ibuf(i)=k
 20         continue

c                       resume the receive task to process the data in ibuf
c           (resume forrecv)
            ids = sys$setef(%val(64))

c                       suspend this task while forrecv processes
c           (suspend forsend)
            ids = sys$waitfr(%val(65))
            ids = sys$clref(%val(65))


 30     continue


        type *,' Program FORSEND is exiting'
c       type *,ibuf,k

c       call sys$delprc(ipid, )

        end
```

```
c  wait.for

        subroutine wait(itim,ids)
        integer*4  sys$waitfr,sys$setimr,sys$bintim
        integer*4  ids,ibintim
        character*(*) itim


        ids=sys$bintim(itim,ibintim)
        if(.not. ids)go to 1000

        ids=sys$setimr(%val(1),ibintim,,)
        if(.not. ids)go to 1000

        ids=sys$waitfr(%val(1))
        if(.not. ids)go to 1000

 1000   return

        end
```

```
c  comglb.for  --  Used to define the installed shareable image
c                  for the global common in some send/receive tests.

        block data

        integer*2 ibuf,iflag
        common/comglb/ibuf(2048),iflag
        data ibuf/2048*0/,iflag/0/
        end
```

```
!  comglb.opt
!  Options file for linking comglb global common
!
dua0:[sys0.syslib]comglb/share
!
```

```
$!  comglbins.com -- Command file to install 'comglb' global common.
$ set noverify
$!  Modules using this common should be linked thus:
$!              $ link module,comglb/opt
$!  where file comglb.opt contains:
$!              comglb/share
$ set verify
$ install delete dua0:[sys0.syslib]comglb
$ install create dua0:[sys0.syslib]comglb.exe;1/share/write
$ ! finished installing comglb
$ set noverify
```

```
$!comglbbld.com --Command file to compile, link, install 'comglb' global common.
$ set noverify
$!  Modules using this common should be linked thus:
$!            $ link module,comglb/opt
$!  where file comglb.opt contains:
$!            comglb/share
$ set verify
$ for comglb
$ link/share comglb
$ purge comglb.obj,comglb.exe
$ install delete dua0:[sys0.syslib]comglb
$ delete dua0:[sys0.syslib]comglb.exe;*
$ copy comglb.exe dua0:[sys0.syslib]comglb.exe;1
$ install create dua0:[sys0.syslib]comglb.exe;1/share/write
$ ! WARNING! Programs using comglb must now be re-linked!
$ ! finished building comglb
$ set noverify
```

```
c  forrecv.for    -  Receive task for 2-task synchronized data transfer.
c                 --  event flag/installed shareable common version

        program forrecv

        integer*2 ibuf,iflag
        integer*2 j,l
        integer*4 ids,sys$ascefc,sys$waitfr,sys$clref,sys$setef

        common/comglb/ibuf(2048),iflag

        ids = sys$ascefc(%val(64),'efcluster',,)        ! assign ef cluster

        j=0


c                       loop forever
 10     continue
            j=j+1

c                       suspend this task until send task has some data
c           (suspend forrecv)
            ids = sys$waitfr(%val(64))

            ids = sys$clref(%val(64))


c                       check data to see if correct

            do 20 l=1,2048
                if( ibuf(l) .ne. j )then
                    type *,'data is incorrect for loop = ',j
                end if
 20         continue


c                       resume the send task to send more data
c           (resume forsend)
            ids = sys$setef(%val(65))

        go to 10



 30     continue

c                       terminate
        end
```

```
c  forsendsave1fil.for
c  forsend.for     --  hiber/wake/'global section-file' version

        program forsend

        integer*2  ibuf,iflag
        integer*2  i,k

        integer*4  ispawn,ids
        integer*4  sys$crmpsc,ipid,chan
        integer*4  inadr(2),retadr(2),secflags

        include  '($secdef)'

        external cli$m_nowait

        common/comglb/ibuf(2048),iflag

        common/ufo/chan
        integer*4  ufo_create
        external   ufo_create

        iflag=0

c                               open the section file

        open( unit=4, file='comglb.tmp',
        1       status='new', initialsize=9,
        1       useropen= ufo_create,err=30)
        close(4)

        secflags = sec$m_gbl .or. sec$m_dzro .or. sec$m_wrt

        inadr(1) = %loc( ibuf(1)     )
        inadr(2) = %loc( iflag       )

c                               create and map to global section

        ids=sys$crmpsc(inadr,retadr,,
        1               %val(secflags),
        1               'glbsec',,,%val(chan)
        1               ,,,,)



c                       start up the receive task to run concurrently
        ispawn=%loc(cli$m_nowait)
        call lib$spawn('run forrecvsave1',,,ispawn,'forrecv',ipid)


c                       -- beginning of forsend

        type *,' The send task (Program FORSEND) is starting'

        do 30 k=1,300

c                       fill buffer
            do 20 i=1,2048
                ibuf(i)=k
 20         continue


c                       resume the receive task to process the data in ibuf
c           (resume forrecv)
            call sys$wake(ipid, )
```

```
c                       suspend this task while forrecv processes
c           (suspend forsend)
            call sys$hiber()

 30     continue

        type *,' Program FORSEND is exiting'

        call sys$delprc(ipid, )

        end
```

```
c  ufo_create.for  -  user-open routine.  Called in 'open' statement

        integer function ufo_create(fab,rab,lun)

        include '($fabdef)'
        include '($rabdef)'

        record /fabdef/fab
        record /rabdef/rab

        integer*4  lun,chan,ids,sys$create

        common /ufo/chan        !common for passing the channel


        fab.fab$l_fop = fab.fab$l_fop .or. fab$m_ufo    !set user-open bit

        ids = sys$create(fab)                           !open the file

        chan = fab.fab$l_stv                            !get channel from fab

        ufo_create = ids                                !set status

        end
```

```
!  glbsec.opt  --  options file for linking the 'mapping' programs
!
!                  this is used to page-align the common block 'comglb'
!
psect_attr = comglb,PAGE
!
```

```
c  forrecvsave1.for
c  forrecv.for  --  hiber/wake/'global section-file' version

        program forrecv

        integer*2 ibuf,iflag
        integer*2 j,l
        integer*4 ids
        integer*4 sys$hiber,sys$wake,sys$crmpsc,sys$mgblsc
        integer*4 inadr(2),retadr(2),secflags

        include  '($secdef)'


        common/comglb/ibuf(2048),iflag

        secflags = sec$m_wrt
        inadr(1) = %loc(ibuf(1))
        inadr(2) = %loc(iflag  )

c                       map to the global section called 'glbsec'

        ids=sys$mgblsc(inadr,retadr,,
        1       %val(secflags),
        1       'glbsec',,)

        j=0

c                       loop forever (we'll stop it with ctrl-y)
 10     continue
        j=j+1

c                       suspend this task until send task has some data
c       (suspend forrecv)
        call sys$hiber()


c                       check data to see if correct

        do 20 l=1,2048
            if( ibuf(l) .ne. j )then
                type *,'data is incorrect for loop = ',j
            end if
 20     continue


c                       resume the send task to send more data
c       (resume forsend)
        call sys$wake( ,'HORNE')

        go to 10


 30     continue

c                       terminate
        end
```

```
c  forsendsave6.for

c  forsend.for  --  shared disk file version

        program forsend

        integer*2  ibuf(2048)
        integer*2  i,k
        integer*4  ispawn,ids
        integer*4  ipid

        integer*2  irecno
        integer*2  j            !loop counter for receive task

        external cli$m_nowait

        common/comglb/irecno,j

        irecno = 0

c                               create and open file
        open(unit=4,file='timetest.dat;1',status='new',
        1       form='unformatted',access='direct',
        1       organization='sequential',recordsize=1024,
        1       initialsize=2800,shared,err=99)


c                               start up the receive task to run concurrently
        ispawn=%loc(cli$m_nowait)
        call lib$spawn('run forrecvsave6',,,ispawn,'forrecv',ipid)

c                               let receive task start
        call lib$wait(%ref(1.0))


c                               -- beginning of forsend

        type *,' The send task (Program FORSEND) is starting'


        do 30 k=1,300           !write 300 records to disk file

c                               fill buffer
            do 20 i=1,2048
                ibuf(i)=k
 20         continue

c                               write to file
            write(4,rec=k,err=98)ibuf
c                               update record number in common
            irecno = irecno+1

 30     continue


                                !wait for recv task to finish
 40     if( j .gt. 300)go to 50
        call lib$wait(%ref(0.01))
        go to 40

 50     type *,' Program FORSEND is exiting'

        call sys$delprc(ipid, )
        go to 60

 98     type *,'forsend: write error on file'
```

```
      go to 60
99    type *,'forsend: open error on file'
60    continue
      end
```

```
c  forrecvsave6.for

c  forrecv.for  --  shared disk file version

        program forrecv

        integer*2 ibuf(2048)
        integer*2 j,l
        integer*4 ids

        integer*2 irecno
        integer*2 itest

        common/comglb/irecno,j

        j=0

c                       open file
        open(unit=4,file='timetest.dat;1',status='old',
        1       form='unformatted',access='direct',
        1       organization='sequential',
        1       recordsize=1024,shared,err=99)


c                       loop forever
 10     continue
        j=j+1           !set j to next record to read

c                       wait for a record to be written
 15     if( irecno .ge. j )then
            go to 16
        else
            call lib$wait(%ref(0.01))
        endif
        go to 15

c                       read file
 16     read(4,rec=j,err=98)ibuf

c                       check data to see if correct

        do 20 l=1,2048
            if( ibuf(l) .ne. j )then
                type *,'data is incorrect for loop = ',j
            end if
 20     continue

        go to 10


 30     continue
        go to 40

 98     type *,'recv: read error, record no. ',j
        go to 40

 99     type *,'recv: open error'

 40     continue

c                       terminate
        end
```

```
/*  csend1.c  send/receive benchmark for UNIX "C" on the Sun  */
/*           This version uses shared memory and signals */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/signal.h>

key_t  keyshm=200;
char *shmaddr=0;
char *shmat();
int shmid;

char crecv[7] = "crecv1";
char *argv;
int func();

int sig2flag=0;

main()
{
    int i, k, pid;
    char *shmbuf;
    short int *ibuf;
    extern int errno;


    printf("create shm seg\n");
                        /* create shared memory segment   */
    if( (shmid = shmget(keyshm,4096,IPC_CREAT|0666)) == -1)
        perror("shmget");

    printf("attach shm seg\n");
                        /* attach shared memory segment   */
    if( (shmbuf = shmat(shmid,shmaddr,0666)) == (char *)(-1))
        perror("shmat");

    ibuf = (short int *)(shmbuf);

                        /* start up the receive task to run concurrently */
    printf("fork the recv process\n");
    if( (pid = fork()) == 0)
        execv(crecv, argv);

    printf("pid= %d\n", pid);
    perror("execv");

    signal(SIGUSR2,func);  /* set up to catch resume signal from recv*/

    sleep(5);

                        /* -- beginning of csend */
    printf( "The send task (Program CSEND) is starting\n");
    for( k=1; k<=300; ++k)
    {
                        /* fill buffer */
        for(i=0; i<2048; ++i)
            ibuf[i] = k;

                        /*resume the receive task to process the data in ibuf*/
        /*(resume crecv)*/
        usleep(100000);    /* delay to make sure recv is paused */
        kill(pid,SIGUSR1);

                        /*suspend this task while crecv processes */
```

```
          /*(suspend csend)*/

/*        printf("send: suspending waiting for recv to check loop %d\n",k);   */
          if( sig2flag != 1 )
              pause();
          sig2flag = 0;
/*        printf("send: resumed, fill buffer for loop %d\n",(k+1));   */
      }
      printf( "program CSEND is exiting\n" );
      kill(pid,9);
}

func(signum, sig_code, scp)
      int signum,sig_code;
      struct sigcontext *scp;
{
      sig2flag = 1;
/*    printf("sigusr2 caught, sig2flag   */
      return;
}
```

```
/*  crecv1.c  send/receive benchmark for UNIX "C" on the Sun  */
/*           This version uses shared memory and signals */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/signal.h>

key_t  keyshm=200;
char *shmaddr=0;
char *shmat();
int shmid;

int sig1flag=0;

main()
{
    int j, m, pid, func();
    char *shmbuf;
    short int *ibuf;


                        /* get shared memory segment   */
    if( (shmid = shmget(keyshm,4096,0666)) == -1)
        perror("shmget");

                        /* attach shared memory segment   */
    if( (shmbuf = shmat(shmid,shmaddr,0666)) == (char *)(-1))
        perror("shmat");

    ibuf = (short int *)(shmbuf);

    pid = getppid();

                        /* loop forever (we'll stop it externally) */

    signal(SIGUSR1,func);  /* set up to catch resume signal from send */

    for(j=1; ; ++j)
    {
                        /* suspend this task until send task has some data*/
        /*(suspend crecv)*/

/*      printf("recv: now suspending waiting for loop %d\n",j);  */
        if( sig1flag != 1 )
            pause();
        sig1flag = 0;
/*      printf("recv: resumed, check data for loop %d\n",j);  */

                        /* check data to see if correct */
        for(m=0; m<2048; ++m)
        {
            if( ibuf[m] != j)
            {
                printf("data is incorrect for loop = %d\n",j);
                printf("     ibuf[%d] = %d\n",m,ibuf[m]);
            }
        }
                        /* resume the send task to send more data */
        /*(resume csend)*/
        usleep(100000);         /* delay to make sure send is paused */
        kill(pid,SIGUSR2);
    }
                        /* terminate */
}
```

```
func(signum, sig_code, scp)
    int signum,sig_code;
    struct sigcontext *scp;
{
    sig1flag = 1;
    return;
}
```

```
/*  csend2.c  send/receive benchmark for UNIX "C" on the Sun  */
/*            This version uses shared memory and semaphores */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

key_t  keyshm=200,  keysem=300;
char *shmaddr=0;
char *shmat();
int shmid,          semid, semval;
struct sembuf   lock0={0,-1,SEM_UNDO};
struct sembuf unlock0={0, 1,SEM_UNDO};
struct sembuf   lock1={0,-1,SEM_UNDO};
struct sembuf unlock1={0, 1,SEM_UNDO};

char crecv[7] = "crecv2";
char *argv;

main()
{
    int i, k, pid;
    char *shmbuf;
    short int *ibuf;
    extern int errno;


    printf("create shm seg\n");
                    /* create shared memory segment  */
    if( (shmid = shmget(keyshm,4096,IPC_CREAT|0666)) == -1)
        perror("send:shmget");

    printf("attach shm seg\n");

                    /* attach shared memory segment  */
    if( (shmbuf = shmat(shmid,shmaddr,0666)) == (char *)(-1))
        perror("send:shmat");

    ibuf = (short int *)(shmbuf);

    if( (semid = semget(keysem,2,0666|IPC_CREAT|IPC_EXCL)) == -1)
        perror("send:semget");

    if( (semval = semctl(semid,0,GETVAL)) == -1)
        perror("send:semctl");
    printf("semval=%d\n",semval);

                    /* start up the receive task to run concurrently */
    printf("fork the recv process\n");
    if( (pid = fork()) == 0)
        execv(crecv, argv);

    printf("pid= %d\n", pid);
    perror("send:execv");

    sleep(5);

                    /* -- beginning of csend */
    printf( "The send task (Program CSEND) is starting\n");
    for( k=1; k<=300; ++k)
    {
                    /* fill buffer */
        for(i=0; i<2048; ++i)
```

```
            ibuf[i] = k;

                        /*resume the receive task to process the data in ibuf*/
        usleep(100000);
        /*(resume crecv)*/
        if( semop(semid,&unlock1,1) == -1)
            perror("send:semop");

        if( (semval = semctl(semid,1,GETVAL)) == -1)
            perror("send:semctl");
/*      printf("send: semval1 after unlock=%d\n",semval);  */


                        /*suspend this task while crecv processes */
        /*(suspend csend)*/
/*      printf("send: suspending waiting for recv to check loop %d\n",k);  */
        if( semop(semid,&lock0,1) == -1)
            perror("send:semop");
/*      printf("send: resumed, fill buffer for loop %d\n",(k+1));  */

        if( (semval = semctl(semid,0,GETVAL)) == -1)
            perror("send:semctl");
/*      printf("send: semval after lock=%d\n",semval);  */



    }
    printf( "program CSEND is exiting\n" );
    kill(pid,9);
    semctl(semid,0,IPC_RMID);
}
```

```
/*  crecv2.c  send/receive benchmark for UNIX "C" on the Sun  */
/*           This version uses shared memory and semaphores */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

key_t  keyshm=200,  keysem=300;
char *shmaddr=0;
char *shmat();
int shmid,          semid, semval;
struct sembuf   lock0={0,-1,SEM_UNDO};
struct sembuf unlock0={0, 1,SEM_UNDO};
struct sembuf   lock1={0,-1,SEM_UNDO};
struct sembuf unlock1={0, 1,SEM_UNDO};

main()
{
    int j, m, pid;
    char *shmbuf;
    short int *ibuf;


    printf("recv: starting recv\n");

                        /* get shared memory segment   */
    if( (shmid = shmget(keyshm,4096,0666)) == -1)
        perror("shmget");

                        /* attach shared memory segment   */
    if( (shmbuf = shmat(shmid,shmaddr,0666)) == (char *)(-1))
        perror("shmat");

    ibuf = (short int *)(shmbuf);

    pid = getppid();

    semid=semget(keysem,2,0);

    semval = semctl(semid,0,GETVAL);
    printf("recv: semval = %d\n",semval);

                        /* loop forever (we'll stop it externally) */
    printf("recv: start recv loop\n");

    for(j=1; ; ++j)
    {
                        /* suspend this task until send task has some data*/
        /*(suspend crecv)*/
/*      printf("recv: now suspending waiting for loop %d\n",j);  */
        semop(semid,&lock1,1);
        semval = semctl(semid,1,GETVAL);
/*      printf("recv: resumed, check data for loop %d\n",j);
        printf("recv: semvall after lock = %d\n",semval);      */

                        /* check data to see if correct */
        for(m=0; m<2048; ++m)  /* check each word of array */
        {
            if( ibuf[m] != j)
            {
                printf("data is incorrect for loop = %d\n",j);
                printf("     ibuf[%d] = %d\n",m,ibuf[m]);
            }
            if( m < 3 || m > 2045)
```

```
                {
/*                  printf("m = %d, ibuf[m] = %d\n",m,ibuf[m]);   */
                }
            }

                                /* resume the send task to send more data */
            usleep(100000);
            /*(resume csend)*/
            semop(semid,&unlock0,1);

            semval = semctl(semid,0,GETVAL);
/*          printf("recv: semval after unlock=%d\n",semval);   */


        }
                                /* terminate */
}
```

```
/*  csend3.c  send/receive benchmark for UNIX "C" on the Sun  */
/*             This version uses shared memory for data and
               a shared memory flag for synchronization */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

key_t  keyshm=200;
char *shmaddr=0;
char *shmat();
int shmid;

char crecv[7] = "crecv3";
char *argv;

main()
{
    int i, k, pid;
    char *shmbuf;
    short int *ibuf;
    extern int errno;


    printf("create shm seg\n");
                        /* create shared memory segment  */
    if( (shmid = shmget(keyshm,8192,IPC_CREAT|0666)) == -1)
        perror("send:shmget");

    printf("attach shm seg\n");

                        /* attach shared memory segment  */
    if( (shmbuf = shmat(shmid,shmaddr,0666)) == (char *)(-1))
        perror("send:shmat");

    ibuf = (short int *)(shmbuf);
    ibuf[2048] = 0;  /* initialize flag for synchronization*/


                        /* start up the receive task to run concurrently */
    printf("fork the recv process\n");
    if( (pid = fork()) == 0)
        execv(crecv, argv);

    printf("pid= %d\n", pid);
    perror("send:execv");

    sleep(5);

                        /* -- beginning of csend */
    printf( "The send task (Program CSEND) is starting\n");

    for( k=1; k<=300; ++k)
    {
                        /* fill buffer */
        for(i=0; i<2048; ++i)
            ibuf[i] = k;

                        /*resume the receive task to process the data in ibuf*/
        /*(resume crecv)*/
/*      printf("send: resume recv to process loop %d\n",k);   */
        ibuf[2048] = 1;


                        /*suspend this task while crecv processes */
            /*(suspend csend)*/
/*          printf("send: suspending waiting for recv to check loop %d\n",k);   */
            while( ibuf[2048] != 0)
                usleep(10000);      /*  delay a minimum amount */
                                /* NOTE: takes the same time for 1 to 10000 usec.*/


/*          printf("send: resumed, fill buffer for loop %d\n",k);   */


        }
        printf( "program CSEND is exiting\n" );
        kill(pid, 9);
}
```

```
/*  crecv3.c  send/receive benchmark for UNIX "C" on the Sun  */
/*           This version uses shared memory for data and
             a shared memory flag for synchronization */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

key_t  keyshm=200;
char *shmaddr=0;
char *shmat();
int shmid;

main()
{
    int j, m, pid;
    char *shmbuf;
    short int *ibuf;


    printf("recv: starting recv\n");

                        /* get shared memory segment  */
    if( (shmid = shmget(keyshm,8192,0666)) == -1)
        perror("shmget");

                        /* attach shared memory segment  */
    if( (shmbuf = shmat(shmid,shmaddr,0666)) == (char *)(-1))
        perror("shmat");

    ibuf = (short int *)(shmbuf);

    pid = getppid();


                        /* loop forever (we'll stop it externally) */
    printf("recv: start recv loop\n");

    for(j=1; ; ++j)
    {
                        /* suspend this task until send task has some data*/
        /*(suspend crecv)*/
/*      printf("recv: now suspending waiting for loop %d\n",j);  */
        while( ibuf[2048] != 1 )
            usleep(10000);     /* delay a minimum amount */
                          /* NOTE: takes the same time for 1 to 10000 usec.*/

/*      printf("recv: resumed, check data for loop %d\n",j);  */

                        /* check data to see if correct */
        for(m=0; m<2048; ++m)  /* check each word of array */
        {
            if( ibuf[m] != j)
            {
                printf("data is incorrect for loop = %d\n",j);
                printf("     ibuf[%d] = %d\n",m,ibuf[m]);
            }
        }

                        /* resume the send task to send more data */
/*      printf("recv: now resume the send process\n");  */
        /*(resume csend)*/
        ibuf[2048] = 0;


    }
```

```
                            /* terminate */
    }
```

```
c  rspns1save1.for
c  rspns1.for  --  Process creation benchmark
c              -- This version uses spawn to create a subprocess

        program rspns1

c!!!!   integer*4       ids, lib$spawn
        character*6     name
        data name/'rspns2'/

        write(6,900)
 900    format(' start response time test ')


        do 10 i=1,100
c                               spawn subprocess and wait for completion

c!!!!       ids = lib$spawn('run rspns2',,,,name)       ! spawn w/run
c!!!!       ids = lib$spawn('rspns2',,,,name)           ! spawn as command
            call  lib$spawn('run rspns2',,,,name)       ! spawn w/run


c!!!!      if(.not. ids) call sysmsg(ids)               ! This code for debug


 10     continue


        write(6,910)
 910    format(' end response time test ')

        end
```

```
c  rspns2.for
        program rspns2
        end
```

```
c  rspns1save5qio.for  -  Process creation timing test for MicroVAX II.

c               This version uses the termination mailbox method to
c               wait for completion of the created process.  The
c               created process is the original version of rspns2.
c               The termination mailbox is read with a QIO call.

        program rspns1

        include         '($prcdef)'
        include         '($dvidef)'
        include         '($iodef)'

        integer*4       ids,sys$creprc,sys$crembx,lib$getdvi,sys$qiow
        integer*4       pid, ichan, mbxunt
        integer*2       itrmsg(512)
        integer*2       iostat(4)
        character*6     name
        data name/'rspns2'/


c                                               create termination mailbox
        ids = sys$crembx( ,ichan,,,,,'mbx')
        if(.not. ids) go to 99

c                                               get device unit no. for mbx
        ids = lib$getdvi(dvi$_unit,ichan,,mbxunt,,)
        if(.not. ids) go to 99


        write(6,900)
 900    format(' start response time test ')


        do 10 i=1,100
c                               create detached process with term. mailbox

            ids = sys$creprc( pid,'dual:[user.horne]rspns2.exe',
        1                       ,,,
        1                       ,,name,%val(4),,
        1                       %val(mbxunt),%val(prc$m_detach) )

            if(.not. ids) go to 99

                                                !read to wait for termination
            ids = sys$qiow( , %val(ichan), %val(io$_readvblk),
        1                       iostat,,,itrmsg,%val(8),,,,)

 10     continue

!!!!    type 901,(itrmsg(k),k=1,42)
!901    format(4(x,10z6.4/),(x,2z6.4))

        write(6,910)
 910    format(' end response time test ')

        if(.not. itrmsg(3))call sysmsg(itrmsg(3))
        go to 100

 99     call sysmsg(ids)                ! This code for debug
 100    continue

        end
```

```
c  timetask1.for  --  hiber/wake version

        program timetask1

        integer*2  i
        integer*4  ispawn,ids
        integer*4  ipid

        integer*4  sys$hiber,sys$wake

        external cli$m_nowait


c                       start up task2 to run concurrently
        ispawn=%loc(cli$m_nowait)
        call lib$spawn('run timetask2',,,ispawn,'timtk2',ipid)

c                       wait while task2 loads
        call lib$wait(%ref(2.0))

        type *,' task1 is starting'

        do 30 i=1,1000


c                       resume task2
c           (resume task2)
            call sys$wake(ipid, )



c                       suspend this task while task2 runs
c           (suspend task1)
            call sys$hiber( )


 30     continue


        type *,' task1 is exiting'

        call sys$delprc(ipid, )
        go to 40

 40     continue
        end
```

```
c  timetask2.for    -- hiber/wake version

        program timetask2

        integer*4 ids
        integer*4 sys$hiber,sys$wake


c                       loop forever
 10     continue

c                       suspend this task until send task1 starts it
c       (suspend task2)
        call sys$hiber( )


c                       resume task1
c       (resume task1)
        call sys$wake( ,'HORNE')


        go to 10


 30     continue

c                       terminate
        end
```

```
/*  proctim.c  --  parent-child process timing test */

#define DELAY 30
int i;
int j;

main()
{
    for(i=1; i<=20; ++i)
    {
        sleep(1);
        if(fork() == 0)
        {
            for(;;)
            {
                printf("i’m task no. %d \n",i);
                sleep(DELAY);
            }
            exit();
        }
    }
}
```

```
c  proctim_sub.for

c  proctim.for -- process timing test, using VAX/VMS subprocesses.
c                       This is the "parent" process, the "child" processes
c                       are copies of proctim1.for.

        program proctim

        integer*4       ids

        include         '($prcdef)'
        integer*4       pid,sys$creprc,lib$spawn
        character*6     prcnam  /'prcn__'/
        external        cli$m_nowait

c******************************************************************************

        do 50 i=1,20

                if( i .lt. 10 )then
                    prcnam(5:5) = '0'
                    prcnam(6:6) = char(48+i)
                else
                    if( i .lt. 20)then
                        prcnam(5:5) = '1'
                        prcnam(6:6) = char(48+i-10)
                    else
                        prcnam(5:5) = '2'
                        prcnam(6:6) = char(48+i-20)
                    endif
                endif


                ids = lib$spawn('run dual:[user.horne]proctim1',,,
        1               %loc(cli$m_nowait),prcnam)


                call lib$wait(%ref(1.0))

 50     continue

        call lib$wait(%ref(200.0))

        end
```

```
c  proctim_det.for

c  proctim.for -- process timing test,  detached process version.
c                       This is the "parent" process, the "child"
c                       processes are copies of proctim1.for.

        program proctim

        integer*4       ids

        include         '($dvidef)'
        character*16    devnam
        integer*2       ilen
        integer*4       lib$getdvi

        include         '($prcdef)'
        integer*4       pid,sys$creprc
        character*6     prcnam  /'prcn__'/

c******************************************************************************

                                !get equivalence name for sys$input device
        ids = lib$getdvi(dvi$_devnam,,'sys$input',,devnam,ilen)

                                !start off 24 "child" processes
        do 50 i=1,24

                if( i .lt. 10 )then
                    prcnam(5:5) = '0'
                    prcnam(6:6) = char(48+i)
                else
                    if( i .lt. 20)then
                        prcnam(5:5) = '1'
                        prcnam(6:6) = char(48+i-10)
                    else
                        prcnam(5:5) = '2'
                        prcnam(6:6) = char(48+i-20)
                    endif
                endif


                                !create process as detached process
                ids = sys$creprc(pid,'dual:[user.horne]proctim1',
        1               devnam(1:ilen),devnam(1:ilen),devnam(1:ilen),
        1               ,,
        1               prcnam,%val(10),,,%val(prc$m_detach))

                call lib$wait(%ref(1.0))

 50     continue

        end
```

```
c  proctiml.for -- "child" process for process timing test,
c                       can be run as a detached process by
c                       proctim_det.for or a subprocess by
c                       proctim_sub.for

        program proctiml

        integer*4       ids

        include         '($jpidef)'
        integer*4       lib$getjpi,jlen
        character*15    prcnam

c******************************************************************************

        ids = lib$getjpi(jpi$_prcnam,,,,prcnam,jlen)

        do 100 i=1,10
            type *,'I''m task no. ',prcnam(5:6)
            call lib$wait(%ref(60.0))
 100    continue

        end
```

```
-- proctim.ada  -  Parent-Child Process timing test, program to start
--                 20 "child" processes, letting them run independently.
--                 Ada version for the MicroVAX II, uses Ada tasks
--                 instead of operating system processes.

with text_io;  use text_io;

procedure proctim is

    idum : integer;

    package my_io is new integer_io( integer );
    use my_io;

--*********************************************************************

                                        -- This is the "child" task. Mul-
                                        -- tiple copies will be started.

    task type test_run is
        entry start;
    end test_run;

    bunch_of_runs : array(1..20) of test_run;   -- declare 20 tasks.


    task body test_run is

        iname : integer := 0;

    begin

        accept start;
        iname := idum;
        for j in 1..5  loop
            put("I'm task ");  put( iname );
            new_line;
            delay 30.0;
        end loop;
    end test_run;

--*********************************************************************

begin

                                        -- This is the "parent" task.
    for i in 1..20  loop
        idum := i;
        bunch_of_runs(i).start;
        delay 1.0;
    end loop;

end proctim;
```

```
--  task1_and_task2.ada  -  an Ada program to test multi-tasking.

--              This program alternates execution between two
--              tasks. There is no synchronization between the
--              two tasks, only delay statements to test
--              alternating execution.

with text_io, calendar;
use text_io, calendar;

procedure task1 is

    package duration_text_io is new fixed_io( duration );
    use duration_text_io;

    t0    : duration;                  -- time for task1.
    t1    : duration;                  -- time for task2.


    task task2;

    task body task2 is

    begin                              -- beginning of task2.

        t1 := seconds( clock );
        put(" task2 starting time ( in seconds from midnight ) = " );
        put( t1 );  new_line;

        delay 5.0;                     -- delay for 5 seconds.

        t1 := seconds( clock );        -- get the time in seconds.
        put(" task2 running, time ( in seconds from midnight ) = " );
        put( t1 );  new_line;          -- print out the current time.

        delay 5.0;                     -- delay for 5 seconds.

        t1 := seconds( clock );        -- get the time in seconds.
        put(" task2 running, time ( in seconds from midnight ) = " );
        put( t1 );  new_line;          -- print out the current time.

        delay 5.0;                     -- delay for 5 seconds.

        t1 := seconds( clock );        -- get the time in seconds.
        put(" task2 running, time ( in seconds from midnight ) = " );
        put( t1 );  new_line;          -- print out the current time.

    end task2;

begin                                  -- beginning of task1.

    t0 := seconds( clock );
    put(" task1 starting time ( in seconds from midnight ) = " );
    put( t0 );  new_line;

    delay 5.0;                         -- delay for 5 seconds.

    t0 := seconds( clock );            -- get the time in seconds.
    put(" task1 running, time ( in seconds from midnight ) = " );
    put( t0 );  new_line;              -- print out the current time.

    delay 5.0;                         -- delay for 5 seconds.

    t0 := seconds( clock );
    put(" task1 running, time ( in seconds from midnight ) = " );
    put( t0 );  new_line;              -- print out the current time.
    delay 5.0;                        -- delay for 5 seconds.

    t0 := seconds( clock );
    put(" task1 running, time ( in seconds from midnight ) = " );
    put( t0 );  new_line;             -- print out the current time.

    delay 5.0;                        -- delay for 5 seconds.

    t0 := seconds( clock );
    put(" task1 running, time ( in seconds from midnight ) = " );
    put( t0 );  new_line;             -- print out the current time.

end task1;
```

```
--  ttqio.ada   -  program to test QIO system service from Ada for
--                 terminal output, MicroVAX II.

with text_io;  use text_io;

with starlet;  use starlet;                     -- used for system calls.
with system;  use system;                       -- used by starlet.
with condition_handling; use condition_handling;-- used for status returns.
with unchecked_conversion;                      -- used for type conversion.

procedure ttqio is

--              The purpose of this function is to convert a value
--              from the type of "address" to the type of
--              "unsigned_longword" so that the value can be used
--              by the  QIO system service routine.

    function convert_addr_to_longword is new unchecked_conversion (
                                        address, unsigned_longword );

--              The purpose of this function is to convert a value
--              from the type of "iosb_type" to the type of
--              "cond_value_type" so that it can be used by the
--              routine sysmsg.

    function convert_status is new unchecked_conversion (
                                        iosb_type, cond_value_type);


                                                -- make declarations necessary
                                                -- to access FORTRAN routine to
                                                -- output system messages.
    procedure sysmsg( ids : cond_value_type );
    pragma interface( fortran,sysmsg );
    pragma import_procedure( sysmsg,mechanism=>reference );


                                                -- declare variables.
    buffer       : constant string := "this is a test";
    adrbuf       : address;
    ttchan       : channel_type;                -- I/O channel.
    ids          : cond_value_type;             -- system service status.
    iostat       : iosb_type;                   -- QIO returned status.
    cond_stat    : cond_value_type;
                                                -- QIO function code.
    ifunc        : function_code_type := io_writepblk;
    qio_p1       : unsigned_longword;           -- QIO buffer address.
    qio_p2       : unsigned_longword := 14;     -- QIO byte count.

begin

    assign( ids,"tt:",ttchan);                  -- call assign system service.
    new_line(2);
    put_line(" assign status:");
    sysmsg( ids );
    new_line(2);

    adrbuf := buffer'address;
    qio_p1 := convert_addr_to_longword ( adrbuf );

                                                -- call QIO system service.
    qiow( status=>ids,chan=>ttchan,func=>ifunc,
          iosb=>iostat,p1=>qio_p1,p2=>qio_p2);

    new_line;
    put_line(" qio status:");
    sysmsg( ids );
    new_line;

    put_line(" i/o status:");
    cond_stat := convert_status ( iostat );
    sysmsg( cond_stat );

end ttqio;
```

```
c  ttqio.for  -  program to test system services for terminal output,
c                MicroVAX II.

        program ttqio

        character*20    buffer
        integer*4       ids,sys$assign,sys$qiow,lib$getjpi,lib$getdvi
        integer*2       iosb(4),ttchan

        character*15    prcnam
        character*16    devnam
        integer*2       ilen
        integer*4       jlen

        data buffer/'this is a test: no.1'/

        include         '($iodef)'
        include         '($ssdef)'

        include         '($jpidef)'
        include         '($dvidef)'

        write(5,900)
 900    format(' ttqio performs 2 qio writes to terminal',/)

        ids=lib$getjpi(jpi$_prcnam,,,,prcnam,jlen)
        type *,'process name = ',prcnam(1:jlen)

        ids=lib$getdvi(dvi$_devnam,,'sys$input',,devnam,ilen)
        type *,'sys$input name = ',devnam(1:ilen)

        call lib$wait(%ref(4.0))

        ids=sys$assign('tt:',ttchan,,)

        type *,' ttchan= ',ttchan,' assign status:'      !"type" tests write
        call sysmsg(ids)                                 !to sys$output
        type *,' '


        ids=sys$qiow(,%val(ttchan),%val(IO$_WRITEPBLK),iosb,,,
        1       %ref(buffer),%val(20),,,,)

        type *,' qio status:'
        call sysmsg(ids)

        type *,'i/o status:'
        call sysmsg(iosb(1))

        call lib$wait(%ref(30.0))                 !wait 10 sec.

        ids=sys$assign('tt:',ttchan,,)

        type *,' ttchan= ',ttchan,' assign status:'
        call sysmsg(ids)
        type *,' '

        buffer(20:20)='2'

        ids=sys$qiow(,%val(ttchan),%val(IO$_WRITEPBLK),iosb,,,
        1       %ref(buffer),%val(20),,,,)

        type *,' qio status:'
        call sysmsg(ids)

        type *,'i/o status:'
```

```
       call sysmsg(iosb(1))

       type *,'ttqio exiting!'
       type 901
901    format(' $ ',$)

       end
```

```
/* ttqio.c  -  Program to test QIO system services from "C"
               for terminal output. NOTE: To link, use the
               following command:
                       link ttqio,sysmsg
               where sysmsg is a FORTRAN subroutine to output
               system messages.                                  */

#include stdio                         /* UNIX 'Standard I/O'
                                          Definitions.           */
#include iodef                         /* I/O Functions Codes
                                          Definitions.           */

#include descrip                       /* VMS Descriptor Definitions. */

char buffer[15] = "this is a test";    /* test output message.   */

int SYS$ASSIGN(),SYS$QIOW();           /* declaration of the system
                                          services (not required).   */

main()
{
    short ttchan,iosb[4];
    int ids,p2=14;

/*-----------------------------------------------------------------*/

                                       /* Information necessary to pass
                                          the argument for the device
                                          name by Descriptor.        */

    struct dsc$descriptor_s  name_desc; /* Name the descriptor       */

    char *name = "tt:";                /* Define device name.        */

                                       /* length of name WITHOUT
                                          null terminator.           */
    name_desc.dsc$w_length = strlen( name );

    name_desc.dsc$a_pointer = name;    /* Put address of shortened
                                          string in descriptor.      */

                                       /* String descriptor class    */
    name_desc.dsc$b_dtype = DSC$K_CLASS_S;

                                       /* Data type: ASCII string    */
    name_desc.dsc$b_dtype = DSC$K_DTYPE_T;

/*-----------------------------------------------------------------*/

                                       /* Call assign system service  */
    ids=SYS$ASSIGN( &name_desc,&ttchan,0,0);

    putchar( '\n' );
    putchar( '\n' );

    printf( "assign status:\n" );
    sysmsg( &ids );

    putchar( '\n' );
    putchar( '\n' );

                                       /* Call QIO system service.   */
    ids=SYS$QIOW( 0,ttchan,IO$_WRITEPBLK,&iosb,0,0,
         &buffer,p2,0,0,0,0 );
```

```
        putchar('\n');
        printf( "qio status:\n" );
        sysmsg( &ids );
        putchar( '\n' );

        ids=iosb[0];
        printf( "i/o status:\n" );
        sysmsg( iosb );

    }
```

```
c  testalarm.for -- program to test alarm call, MicroVAX II.
c                    This program uses an event flag to signal
c                    an alarm after a specified period of time.

        program testalarm

        real*4          dtime
        integer*4       delta(2)
        integer*4       istat,ival,iflag,itimr,sys$readef

        include         '($ssdef)'

        iflag = 1
        itimr = 1

        type *,'enter alarm delay time, in seconds (real number)'
        accept *,dtime

        type *,'calling alarm routine'
                                                !Convert delay time to VMS
        call rtimsysbin(dtime,delta)            !quadword format.
                                                !Set timer.
        call sys$setimr(%val(iflag),delta,,%val(itimr))

        type *,'waiting for event flag'

 10     istat = sys$readef(%val(iflag),ival)
        if( istat .eq. SS$_WASSET )go to 20
        call lib$wait(%ref(0.01))
        go to 10
 20     continue

        type *,'ALARM!'

        end
```

```
c  testalarm_ast.for -- program to test alarm call, MicroVAX II.

c       This program uses an AST routine to interrupt the main program
c       and signal an alarm after a specified period of time.

        program testalarm

        real*4          dtime
        integer*4       delta(2)
        integer*4       istat,ival,itimr

        external        alarmast                    !must use else a reserved opcode
                                                    !fault occurs
        common/astcom/istat

        include         '($ssdef)'

        itimr = 1
        istat = 0

        type *,'enter alarm delay time, in seconds (real number)'
        accept *,dtime


                                                    !Convert delay time to VMS
        call rtimsysbin(dtime,delta)                !quadword format.

                                                    !call system routine to
                                                    !set timer and specify
                                                    !the AST routine.
        call sys$setimr(,delta,alarmast,%val(itimr))

        type *,'waiting for alarm AST'

 10     continue
        if( istat .eq. SS$_WASSET )go to 20
        call lib$wait(%ref(0.01))
        go to 10
 20     continue

        type *,'ALARM!'

        end



        subroutine alarmast                         !this is the AST routine.

        include         '($ssdef)'
        integer*4       istat
        common/astcom/istat

        istat = SS$_WASSET

        return
        end
```

```
c  rtimsysbin.for  --  will convert an amount of time (in seconds,
c                        to a resolution of 0.01 sec), to system
c                        quadword format.

        subroutine rtimsysbin(tim,deltim)

        real*4          tim             !time, in seconds, real number
        integer*4       itics,idays,ihrs,imin,isec,ihsec,irem
        character*16    string
        integer*4       deltim(2)       !time, in vax system quadword format
c******************************************************************************
        itics = tim*100.                !convert "tim" to integer number of
                                        ! 0.01 sec. ticks

        idays = itics/8640000           !(100*60*60*24)ticks/day
        irem = jmod(itics,8640000)

        ihrs = irem/360000              !(100*60*60)ticks/hour
        irem = jmod(irem,360000)

        imin = irem/6000                !(100*60)ticks/min
        irem = jmod(irem,6000)

        isec = irem/100                 !(100)ticks/sec
        ihsec = jmod(irem,100)          !ticks

        write(string,900)idays,ihrs,imin,isec,ihsec
 900    format(i4,' ',i2.2,':',i2.2,':',i2.2,'.',i2.2)

        type 900,idays,ihrs,imin,isec,ihsec     !DEBUG

        call sys$bintim(string,deltim)  !convert from string to system delta
                                        !format

        return
        end
```

```
--  Testalarm_ast.ada  -  a program to test alarm call, MicroVAX II.

--      This program uses an AST routine to interrupt the main program
--      and signal an alarm after a specified period of time.

--      NOTE: The link command for this program is as follows:
--                 acs link testalarm_ast rtimsysbin


with text_io;  use text_io;
with starlet;  use starlet;                     -- used for system calls.
with system;  use system;                       -- used by starlet.
with condition_handling; use condition_handling;-- used for status returns.

procedure testalarm_ast is

    package my_io is new float_io( float );
    use my_io;

    procedure rtimsysbin( dtime : float; delta_tim : date_time_type );
    pragma interface( fortran,rtimsysbin );
    pragma import_procedure( rtimsysbin, mechanism=>reference );

    dtime     : float := 0.0;                   -- input delay time.
    ids       : cond_value_type;                -- system service status.
    delta_tim : date_time_type;                 -- delay time (VMS quadword).
    istat     : integer := 0;
    itimr     : unsigned_longword := 1;

--*********************************************************************
    task handler is                             -- THIS IS THE AST ROUTINE.
        entry receive_ast;
        pragma ast_entry( receive_ast );
    end handler;

    task body handler is
    begin
        accept receive_ast;
        istat := SS_WASSET;
    end handler;
--*********************************************************************

begin

    loop

        begin

            put_line("enter alarm delay time, in seconds (real number)");
            new_line;

            get( dtime );                       -- get the input delay time.
            exit;                               -- exit loop if no error.

        exception
            when data_error =>
                skip_line; put("input error - try again");  new_line( 2 );

        end;

    end loop;

                                                -- call fortran subroutine to
                                                -- convert delay time to VMS
                                                -- quadword format.
    rtimsysbin( dtime,delta_tim );

                                                -- call system routine to
                                                -- set the timer and to
                                                -- specify the AST routine.

    setimr( status=>ids, daytim=>delta_tim,
            astadr=>handler.receive_ast'ast_entry,
            reqidt=>itimr );

    put_line("waiting for alarm AST");  new_line;

    loop                                        -- loop forever.
        delay 0.01;
        if istat = SS_WASSET  then
            exit;                               -- exit the loop.
        end if;
    end loop;

    put( "ALARM!" );  new_line;

end testalarm_ast;
```

```
/*  testalarm_ast.c -- program to test alarm call, MicroVAX II.        */

/*     This program uses an AST routine to interrupt the main program */
/*     and signal an alarm after a specified period of time.          */

#include ssdef                            /* used for system services    */


int istat;

main()
{
    extern alarmast();

    float dtime;
    int delta[2],ival,itimr;


    itimr =1;
    istat = 0;

    printf("enter alarm delay time, in seconds (real number)\n");
    scanf("%f",&dtime);

    rtimsysbin(&dtime,delta);

                                         /* call system routine to set */
                                         /* the timer and to specify   */
                                         /* the AST routine.           */
    sys$setimr(0,delta,alarmast,itimr);

    printf("waiting for alarm AST\n");

    while( istat != SS$_WASSET )
    {
        lib$wait(0.01);
    }

    printf("ALARM\n");

}

/*
alarmast()                               /* This is the AST routine.    */
{

    istat = SS$_WASSET;

}
*/
```

```
--  adadelay.ada  -  An Ada program to test the Ada "delay" statment,
--                   MicroVAX II.

with calendar, text_io;
use calendar, text_io;

procedure adadelay is

    package duration_io is new fixed_io( duration );
    use duration_io;

    interval  : constant duration := 10.0;
    t1        : duration;
    t2        : duration;
    dtime     : duration;
    ptime     : duration;

begin

    t1 := seconds( clock );

    loop

        t2 := seconds( clock );

        ptime := t2 - t1;
        dtime := (interval - ( ptime ) );
        delay dtime;

        t1 := seconds( clock );

        put(" proc. time = " );  put ( ptime );  new_line;
        put(" delay time = ");  put( dtime );  new_line;
        put(" total time = ");  put( dtime+ptime ); new_line; new_line;

    end loop;

end;
```

```
-- task_order.ada  - An Ada program used to determine the order of
--                   activation of multiple tasks, MicroVAX-II.

with text_io;  use text_io;

procedure task_order is

--*********************************************************************


    task task4;
    task task3;
    task task2;
    task task1;

--*********************************************************************

    task body task2 is

    begin                              -- beginning of task2.

        put(" Task2 is starting");  new_line;

    end task2;

--*********************************************************************
--*********************************************************************

    task body task1 is

    begin                              -- beginning of task1.

        put(" Task1 is starting");  new_line;

    end task1;

--*********************************************************************
--*********************************************************************

    task body task4 is

    begin                              -- beginning of task4.

        put(" Task4 is starting");  new_line;

    end task4;

--*********************************************************************
--*********************************************************************

    task body task3 is

    begin                              -- beginning of task3.

        put(" Task3 is starting");  new_line;

    end task3;

--*********************************************************************

begin                                  -- beginning of task_order.

    put(" The driver (task_order) is starting");  new_line;
```

```
end task_order;
```

```
-- task_exec.ada  -  Ada multi-task time-shared execution test.

--                      This is a test program to run three tasks
--                      by alternating their execution, letting
--                      each one run for a specified time interval.
--                      (MicroVAX-II version.)
--              NOTE:   This program is designed to run without
--                      "time-slicing" of any kind.

with text_io;  use text_io;
with calendar;  use calendar;

procedure task_exec is

    package duration_io is new fixed_io( duration );
    use duration_io;

    package my_io is new integer_io( integer );
    use my_io;

--********************************************************************
--  procedure check -- reentrant procedure to let calling task
--                     control its own execution.

--  This procedure checks to see if the process time has expired.
--  THIS PROCEDURE IS USED INSTEAD OF A TIME_SLICE MECHANISM.

    procedure check( interval : in duration; t1 : in out duration ) is

        t2    : duration;                   -- current time (seconds).
        dtime : duration;                   -- delta time (seconds).

    begin
        t2 := seconds( clock );             -- get curent time (seconds).
        dtime := t2 - t1;                   -- calculate how long the calling
                                            -- task has been running.
        if dtime >= interval  then

            delay 0.01;                     -- delay just enough to schedule
                                            -- another task.
--                .
--                .                         -- (The calling task delays here
--                .                         -- while other tasks run.)
--                .

            t1 := seconds( clock );         -- get new start time (seconds).

        end if;
    end check;
--********************************************************************

--********************************************************************

    task task3;             -- declare tasks such that task1 starts first.
    task task2;
    task task1;

--********************************************************************

    task body task1 is

        t1       : duration;                -- process start time (seconds).
        interval : constant duration := 5.0;  -- desired process time.
        icnt     : integer := 0;

    begin                                   -- beginning of task1.
```

```
        put(" Task1 is starting");  new_line;
        tl := seconds( clock );         -- set starting time.
        loop                            -- loop forever.
            check( interval,tl );       -- see if time has elapsed.
            icnt := icnt + 1;
            put("Task1 = ");  put(icnt);
            put("                                  ");  new_line;
        end loop;
    end task1;

--**********************************************************************
--**********************************************************************

    task body task2 is

        tl       : duration;            -- process start time (seconds).
        interval : constant duration := 5.0;  -- desired process time.
        icnt     : integer := 0;

    begin                               -- beginning of task2.
        put(" Task2 is starting");  new_line;
        tl := seconds( clock );         -- set starting time.
        loop
            check( interval,tl );       -- see if time has elapsed.
            icnt := icnt + 1;
            put("                Task2 = ");  put(icnt);
            put("                ");  new_line;
        end loop;
    end task2;

--**********************************************************************
--**********************************************************************

    task body task3 is

        tl       : duration;            -- process start time (seconds).
        interval : constant duration := 5.0;  -- desired process time.
        icnt     : integer := 0;

    begin                               -- beginning of task3.
        put(" Task3 is starting");  new_line;
        tl := seconds( clock );         -- set starting time.
        loop                            -- loop forever.
            check( interval,tl );       -- see if time has elapsed.
            icnt := icnt + 1;
            put("                               Task3 = ");  put(icnt);
            new_line;
        end loop;
    end task3;

--**********************************************************************
begin                                   -- beginning of task_exec.

    put(" task_exec is starting");  new_line;

end task_exec;
```

```
-- tstslice.ada  -  Multi-task time-shared execution program using
--                  time-slicing, IBM PC AT version.

--                  This program is designed to perform the same functions
--                  as tsk_exec.ada except using 5-second time-slicing
--                  instead of the reentrant check subprogram.

--                  To bind with 5-second time-slicing, use the following
--                  command:

--                      bind tstslice,adalib,options=(slice=5000)


with text_io;  use text_io;

procedure tstslice is

--************************************************************
    task task1;
    task task2;
    task task3;
--************************************************************
    task body task1 is

    begin
        put("task1 is starting");  new_line;
        loop                                -- loop forever.
            put("Task1");  new_line;
        end loop;
    end task1;
--*************************************************************
    task body task2 is

    begin
        put("Task2 is starting");  new_line;
        loop
            put("          Task2");  new_line;
        end loop;
    end task2;
--*************************************************************
    task body task3 is

    begin
        put("Task3 is starting");  new_line;
        loop
            put("                    Task3");  new_line;
        end loop;
    end task3;
--*************************************************************
begin
    put(" tstdelay is starting");  new_line;
end tstslice;
```

```
-- main.ada  - program to test calling a FORTRAN function (innerprod)
--             from an Ada program, MicroVAX II.

--              NOTE: Use the following command to link:
--                      acs link main innerprod

with text_io;  use text_io;

procedure main is

    package new_float is new float_io( float );
    use new_float;

    type array1 is array( integer range <> ) of float;

    function innerprod( a,b : array1; n : integer ) return float;
    pragma interface( fortran,innerprod );
    pragma import_function( innerprod, mechanism => reference );

    q  : array1( 1..100 ) := ( 1..100 => 1.0 );
    t  : array1( 1..100 ) := ( 1..100 => 1.0 );
    p  : float;

begin

    p := innerprod( q, t, q'length );

    put( "returned value = " );  put( p );  new_line;

end main;
```

```
c  innerprod.for  -  FORTRAN function, called by an Ada
c                    program ( main.ada ), MicroVAX II.

      function innerprod( a,b,n )

c     This routine multiples two one-dimensional arrays.
c     element-by-element, then sums the products.

c     Declare A and B as arrays of real numbers.

      real innerprod, a( n ), b( n )

      sum = 0.0

      do 100 i = 1,n
          sum = sum + a( i ) * b( i )
  100 continue

      innerprod = sum

      return
      end
```

```
-- main1.ada  -  program to test calling a FORTRAN subroutine (innerprod1)
--               from an Ada program, MicroVAX II.

--              NOTE: Use the following command to link:
--                      acs link main1 innerprod1

with text_io;  use text_io;

procedure main1 is

    package new_float is new float_io( float );
    use new_float;

    type array1 is array( integer range <> ) of float;

    procedure innerprod1( a,b : array1; n : integer; sum : in out float );
    pragma interface( fortran,innerprod1 );
    pragma import_procedure( innerprod1, mechanism => reference );


    q  : array1( 1..100 ) := ( 1..100 => 1.0 );
    t  : array1( 1..100 ) := ( 1..100 => 1.0 );
    p  : float;

begin

    innerprod1( q, t, q'length, p );

    put( "returned value = " );  put( p );  new_line;

end main1;
```

```
c  innerprod1.for   -  FORTRAN subroutine, called by an Ada
c                      program (main1.ada), MicroVAX II.

      subroutine innerprod1( a,b,n,sum )

c     This routine multiples two one-dimensional arrays.
c     element-by-element, then sums the products.

c     Declare A and B as arrays of real numbers.

      real a( n ), b( n )

      sum = 0.0

      do 100 i = 1,n
          sum = sum + a( i ) * b( i )
  100 continue

      return
      end
```

```
c  formain.for  -  program to test calling an Ada function (nfind)
c                  from a FORTRAN subroutine, MicroVAX II.

c               NOTE: Use one of the the following commands to link:
c                       acs link/nomain nfind formain, OR
c                       link formain,[xxx]nfind
c
c                         where [xxx] is the directory of the ada
c                                     library that contains the object
c                                     file for "nfind".

       program formain

       character*12 x
       character*1  b

       x = '1234 6789'
       b = ' '

       n = nfind( x, %ref( b ) )

c                                      The %ref mechanism specifier causes
c                                      b to be passed by reference.

       type *, b , n

       end
```

```
--  nfind.ada  --  An Ada function, called by a FORTRAN program
--                 ( formain), MicroVAX II.

function nfind( str : string;
                c   : character ) return integer is

begin

    for i in str’range  loop

        if str( i ) = c  then          -- a match was found.
            return i;
        end if;

    end loop;

    return 0;                          -- there was no match.

end;

pragma export_function( nfind );
```

```
-- main2.ada  -  program to test calling a FORTRAN subroutine, which
--                then calls a VAX system service, from an Ada program.
--                MicroVAX II.

with text_io;  use text_io;

procedure main2 is

    package new_integer is new integer_io( integer );
    use new_integer;

    procedure sysmsg( msgnum : integer );
    pragma interface( fortran,sysmsg );
    pragma import_procedure( sysmsg, mechanism => reference );

    imsg : integer;

begin

    put(" Please enter system error number: ");

    get( imsg );
    new_line;  new_line;
    sysmsg( imsg );

end main2;
```

```
c  sysmsg.for

c  This routine writes out the appropriate system message, then returns
c  to the caller instead of killing the program like lib$signal does.

        subroutine sysmsg(ids)

        integer*4  ids,sys$getmsg
        integer*2 msglen
        character*256 msgbuf


                                !call system service to get system
                                !error message from status value.
        istat=sys$getmsg(%val(ids),msglen,msgbuf,%val(15), )
        if(.not. istat)go to 100


        type 900,msgbuf(1:msglen)
 900    format(x,a<msglen>)

 100    return

        end
```

```
--  main1.ada  -  Version for the SUN 3/260.

--  NOTE: To link, use the following command:

--     bind main1,adalib,interface=(modules="innerprod1.o",search="/lib/libc.a")

with text_io;  use text_io;

procedure main1 is

    package new_float is new float_io( float );
    use new_float;

    type array1 is array( integer range <> ) of float;
    procedure innerprod1(a,b : array1; n : in out integer; sum : in out float);
    pragma interface( fortran,innerprod1 );
    pragma interface_name( innerprod1, "innerprod1_");

    q : array1( 1..100 ) := ( 1..100 => 1.0 );
    t : array1( 1..100 ) := ( 1..100 => 1.0 );
    p : float;
    r : integer;

begin

    r := q'length;

    innerprod1( q, t, r, p );

    put( " returned value = " );  put( p );  new_line;

end main1;
```

```
c  innerprodl.f  -  Version for the SUN 3/260.

      subroutine innerprodl( a,b,n,sum )

c     This routine multiples two one-dimensional arrays,
c     element-by-element, then sums the products.

c     Declare A and B as arrays of real numbers.

      integer*2 n
      real a( n ), b( n )

      sum = 0.0

      do 100 i = 1,n
      sum = sum + a( i ) * b( i )
 100  continue

      return
      end
```

```
--   main2.ada  -  Version for the SUN 3/260.

--   NOTE: To link, use the following command:

--      bind main2,adalib,interface=(modules="innerprod.o",search="/lib/libc.a")

with system;
with text_io;  use text_io;

procedure main2 is

    package new_float is new float_io( float );
    use new_float;

    type array1 is array( integer range <> ) of float;

    function innerprod( a,b : array1; n : system.address ) return float;
    pragma interface( fortran,innerprod );
    pragma interface_name( innerprod,"innerprod_" );

    val_n : integer;
    q     : array1( 1..100 ) := ( 1..100 => 1.0 );
    t     : array1( 1..100 ) := ( 1..100 => 1.0 );
    p     : float;

begin

    val_n := q’length;

    p := innerprod( q, t, val_n’address );

    put( " returned value = " );  put( p );  new_line;

end main2;
```

```
c  innerprod.f  -  Version for the SUN 3/260.

        function innerprod( a,b,n )

c       This routine multiples two one-dimensional arrays,
c       element-by-element, then sums the products. Declare
c       A and B as arrays of real numbers.

        integer*2 n
        real innerprod, a( n ), b( n )

        sum = 0.0

        do 100 i = 1,n
        sum = sum + a( i ) * b( i )
  100   continue

        innerprod = sum

        return
        end
```

```
-- adafork.ada  --  This version is for the SUN 3/260.

--  Note: Use the following command to link:
--      bind adafork,adalib,interface=(search="/lib/libc.a")

with system;
with calendar;  use calendar;
with text_io;  use text_io;

procedure adafork is

    package my_io is new integer_io( integer );
    use my_io;

    function fork return integer;
    pragma interface( c,fork );

    function sleep (param1: integer) return integer;
    pragma interface( c, sleep);

    i     : integer;
    stat  : integer := 1;
    dtime : duration := 30.0;

begin --adafork

    fork_loop: for i in 1..20  loop
--      delay 1.0;
        stat := sleep(1);
        stat := fork;
        if stat = 0  then
            for j in 1..5  loop
                put(" i'm task no. "); put( i );  new_line;
                delay dtime;
            end loop;
            exit fork_loop;
        end if;
    end loop fork_loop;

end adafork;
```

```
-- char.ada    -  Ada program used to perform an asychronous, two-task
--                I/O loading analysis, MicroVAX II version.

with text_io;  use text_io;
with calendar;  use calendar;

procedure char is                   -- char is the driver for the two tasks.

    pragma time_slice ( 5.0 );

    task pra is
        entry a;
    end pra;

    task prb is
        entry b;
    end prb;

    task body pra is
    begin
        accept a do
            loop                    -- loop forever.
                put( "A" );
             end loop;
        end a;
    end pra;

    task body prb is
    begin
        accept b do
            delay 1.0;
            loop                    -- loop forever.
                put( "B" );
            end loop;
        end b;
    end prb;

begin                               -- driver justs starts the two tasks.

    prb.b;
    pra.a;

end char;
```

```
-- procload.ada  -  An Ada program used to perform a multiple process
--                  loading analysis, MicroVAX II.

with text_io;  use text_io;

procedure procload is

    pragma time_slice ( 0.01 );          -- 10 milliseconds.

    idum : integer;

    package my_io is new integer_io( integer );
    use my_io;

    flags:array (integer range 0..8190) of boolean;
    i,prime,k,count,iter:integer;

--**********************************************************************

    task type test_run;

    type run_name is access test_run;
    run1 : run_name;

    task body test_run is

        iname : integer := 0;

    begin

        iname := idum;

        for j in 1..10  loop
            put("I'm task ");  put( iname );
            new_line;
            delay 30.0;
        end loop;
    end test_run;

--**********************************************************************

begin

    for i in 1..20  loop
        idum := i;
        run1 := new test_run;
        delay 1.0;
    end loop;
    delay 60.0;

--                               code to cause a computational load
--                               on the system (zprime.ada code).

    put("1550 iterations"); new_line;

    for iter in 1..1550 loop
        count := 0;

        flags := ( 0..8190 => TRUE );

        for i in 0..8190 loop
            if flags(i) then
                prime := i + i + 3;
                k := i + prime;
                while k <= 8190 loop
                    flags(k) := FALSE;
                    k := k + prime;
                end loop;

                count := count + 1;
            end if;
        end loop;

    end loop;

    put(count);put(" primes"); new_line;
--**********************************************************************

end procload;
```